PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

**(51) International Patent Classification 6 :** G06F 9/46 — A1

**(11) International Publication Number:** WO 98/12635

**(43) International Publication Date:** 26 March 1998 (26.03.98)

**(21) International Application Number:** PCT/US97/16511

**(22) International Filing Date:** 16 September 1997 (16.09.97)

**(30) Priority Data:**
60/026,573 17 September 1996 (17.09.96) US

**(60) Parent Application or Grant**
(63) Related by Continuation
US 60/026,573 (CIP)
Filed on 17 September 1996 (17.09.96)

**(71) Applicant** *(for all designated States except US)*: RADISYS CORPORATION [US/US]; 5445 N.E. Dawson Creek Parkway, Hillsboro, OR 97124 (US).

**(72) Inventors; and**
**(75) Inventors/Applicants** *(for US only)*: GERBER, Richard, P. [US/US]; 7458 S.W. Daisy Avenue, Aloha, OR 97007 (US). PETERSON, Scott, D. [US/US]; 6909 S.W. 54th Avenue, Portland, OR 97219 (US).

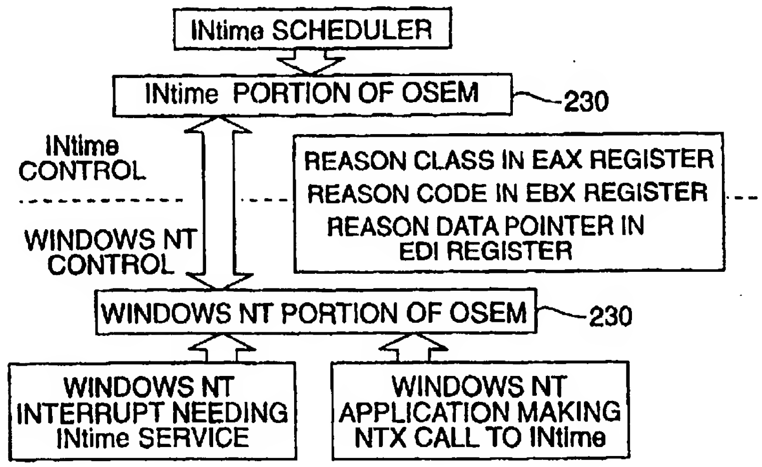
**(74) Agent:** SCHEINBERG, Michael, O.; Stoel Rives LLP, Suite 2300, 900 S.W. Fifth Avenue, Portland, OR 97204-1268 (US).

**(81) Designated States:** AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

**Published**
*With international search report.*
*Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.*

**(54) Title:** METHOD AND APPARATUS FOR ENCAPSULATING A PROTECTED-MODE OPERATING SYSTEM WITHIN A REAL-TIME, PROTECTED-MODE OPERATING SYSTEM

**(57) Abstract**


INtime SCHEDULER
INtime PORTION OF OSEM
230
INtime CONTROL
WINDOWS NT CONTROL
REASON CLASS IN EAX REGISTER
REASON CODE IN EBX REGISTER
REASON DATA POINTER IN EDI REGISTER
WINDOWS NT PORTION OF OSEM
230
WINDOWS NT INTERRUPT NEEDING INtime SERVICE
WINDOWS NT APPLICATION MAKING NTX CALL TO INtime

The present invention is a method of encapsulating an operating system that fully implements protected mode within a second operating system that also fully implements protected mode. In a preferred embodiment, the Windows NT operating system is encapsulated within the INtime real-time operating system, thereby providing a user with a powerful GUI to use when developing and running applications that require a bounded interrupt latency. An encapsulation application (94) running under Windows NT uses a protection level 0 driver (88) to load a boot image (96) of the INtime operating system. The driver saves in global variables the addresses of the Windows NT GDT (272) and IDT (260), then loads pointers to the INtime GDT (258) and IDT (260) into the appropriate CPU registers (12), and causes a jump to the INtime operating system, which begins its initialization. INtime runs an encapsulation subsystem (114) that prepares a single task in which Windows NT is run in under INtime. When the user requests the use of Windows NT or when NT services are required, the encapsulation subsystem causes a switch back to the Windows NT task, which continues to run the driver. The driver then reloads the Windows NT GDT and IDT. Windows NT is therefore unaware that it is running as a task in another operating system. The GDT and IDT from Windows NT contain no information about INtime. The INtime GDT, on the other hand, contains descriptors of the segments used by Windows NT and its IDT, and INtime can control the operation of both operating systems to ensure a predictable response to real-time processes. By writing codes and a data pointer into registers, messages are passed between operating systems and real-time extensions to Windows NT.

*FOR THE PURPOSES OF INFORMATION ONLY*

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

| | |
|---|---|
| AL | Albania |
| AM | Armenia |
| AT | Austria |
| AU | Australia |
| AZ | Azerbaijan |
| BA | Bosnia and Herzegovina |
| BB | Barbados |
| BE | Belgium |
| BF | Burkina Faso |
| BG | Bulgaria |
| BJ | Benin |
| BR | Brazil |
| BY | Belarus |
| CA | Canada |
| CF | Central African Republic |
| CG | Congo |
| CH | Switzerland |
| CI | Côte d'Ivoire |
| CM | Cameroon |
| CN | China |
| CU | Cuba |
| CZ | Czech Republic |
| DE | Germany |
| DK | Denmark |
| EE | Estonia |
| ES | Spain |
| FI | Finland |
| FR | France |
| GA | Gabon |
| GB | United Kingdom |
| GE | Georgia |
| GH | Ghana |
| GN | Guinea |
| GR | Greece |
| HU | Hungary |
| IE | Ireland |
| IL | Israel |
| IS | Iceland |
| IT | Italy |
| JP | Japan |
| KE | Kenya |
| KG | Kyrgyzstan |
| KP | Democratic People's Republic of Korea |
| KR | Republic of Korea |
| KZ | Kazakstan |
| LC | Saint Lucia |
| LI | Liechtenstein |
| LK | Sri Lanka |
| LR | Liberia |
| LS | Lesotho |
| LT | Lithuania |
| LU | Luxembourg |
| LV | Latvia |
| MC | Monaco |
| MD | Republic of Moldova |
| MG | Madagascar |
| MK | The former Yugoslav Republic of Macedonia |
| ML | Mali |
| MN | Mongolia |
| MR | Mauritania |
| MW | Malawi |
| MX | Mexico |
| NE | Niger |
| NL | Netherlands |
| NO | Norway |
| NZ | New Zealand |
| PL | Poland |
| PT | Portugal |
| RO | Romania |
| RU | Russian Federation |
| SD | Sudan |
| SE | Sweden |
| SG | Singapore |
| SI | Slovenia |
| SK | Slovakia |
| SN | Senegal |
| SZ | Swaziland |
| TD | Chad |
| TG | Togo |
| TJ | Tajikistan |
| TM | Turkmenistan |
| TR | Turkey |
| TT | Trinidad and Tobago |
| UA | Ukraine |
| UG | Uganda |
| US | United States of America |
| UZ | Uzbekistan |
| VN | Viet Nam |
| YU | Yugoslavia |
| ZW | Zimbabwe |

# Method and Apparatus for Encapsulating a Protected-Mode Operating System Within a Real-Time, Protected-Mode Operating System

## Technical Field

This application relates to a real-time, fully protected-mode operating system that encapsulates another fully protected-mode operating system to provide users and software developers with a familiar graphical user interface ("GUI") environment together with the predictable response time of a real-time operating system.

## Background of the Invention

Computer systems designed to control real-world activities such as manufacturing processes or traffic flow must perform computing operations within predetermined time constraints governed by the physical process. Such systems cannot let a manufacturing operation produce defective goods or even endanger workers because the system was busy painting an attractive user interface on the screen. Systems that work within such time constraints are called "real-time" systems, and the applications they run are "real-time" applications, as opposed to "run-time" systems and applications. Although "real time" is often interpreted to mean fast, a more appropriate description would be predictable or deterministic. Real-time systems must respond to external, asynchronous events within a predictable time frame and, therefore, must support asynchronous input/output ("I/O"). This support goes further than just making I/O fast; it must enable a system to concurrently execute other portions of an application during I/O operations.

Real-time applications have long been restricted to running on expensive proprietary hardware, but personal computers are attractive alternatives to such hardware. Personal computers are relatively inexpensive, support a range of GUIs, and have a variety of software packages available. Unfortunately, popular run-time operating systems, such as DOS, Windows, Windows 95, Windows NT, and OS/2, used on personal computers cannot guarantee a limit on the interrupt latency, the elapsed time from when an interrupt occurs to when its interrupt handler starts to execute. Because users are familiar with the GUI available in popular run-time operating systems, it would be desirable to provide such an interface to users and software developers while providing real-time operating capability for controlling real-world processes.

Most modern personal computers are based upon a central processing unit ("CPU") using the "Intel Architecture." The Intel Architecture, also known as X86 or 80X86 architecture, is embodied in microprocessors such as the 80386, 80486, Pentium, and Pentium Pro manufactured by Intel Corporation, as well as similar microprocessors manufactured by Advanced Micro Devices, Cyrix, NexGen, and others. The structure of the Intel Architecture supports multitasking, that is, allowing more than one task to operate concurrently. The Intel Architecture also supports protected-mode addressing, that is, assigning protection levels from 3 to 0 to different segments of memory so that applications cannot interfere with each other or with the basic functioning of the microprocessor. Protection level 3 is the least privileged level and is typically used by applications. Protection level 0 is the most privileged level and is typically reserved for use by the operating system and kernel-mode device drivers.

The implementation of multitasking and protected-mode operation in the Intel Architecture has made it difficult to combine on a single personal computer a real-time operating system with a popular run-time operating system having a strong GUI. One difficulty in combining real-time operating systems with run-time operating systems has been providing the real-time operating system with the ability to preempt the run-time operating system when the real-time operating system needs control of the CPU to perform a task within its required time limit.

Fig. 1 is a block diagram providing a simplified overview of a portion of the Intel Architecture. A CPU 10 has a number of internal registers 12 that are used to manage memory, keep track of the system state, set various parameters, and store instructions and information. A main memory 14, which is typically not positioned on the same chip as CPU 10, stores data and instructions. The Intel Architecture is able to address a very large memory space in protected mode by dividing the addressing space into segments and, optionally, into pages. Various tables stored in main memory 14 are used to organize the segments and pages in main memory 14. To facilitate multitasking, the Intel Architecture maintains memory structures that keep track of the state of multiple tasks running concurrently.

Memory is organized and accessed using a global descriptor table ("GDT"). Fig. 2 shows an excerpt from a typical GDT 16. Each entry in GDT 16 is a descriptor 18. There are several types of descriptors 18. Segment descriptors 20 describe a segment of memory by providing its base address and its extent or limit. A memory address is determined by locating a segment base address and then adding the content of an index register. That is, the content of the index register defines an offset from the segment base. The combination of segment base and index is known as the linear address. Each segment descriptor also includes additional information, such as the protection level of the descriptor, whether the segment is a system or an application segment, and whether the segment is currently stored in memory.

The GDT also includes gate descriptors 30, which control access to code and data sections by programs having a higher protection level (lower privilege level) than that of the segment accessed. There can be only one GDT, and its address is stored in a CPU register 32 (Fig. 3) called the global descriptor table register ("GDTR").

Individual tasks can create and use a local descriptor table ("LDT") to keep track of memory segments that are used by the task. Entries in the LDT are in the same format as entries in the GDT. An entry in the LDT describes the base address and limit of each LDT segment. A local descriptor table register ("LDTR") 34 contains a selector that points to the entry in the GDT that describes any currently active LDT.

A linear address is determined by using a "selector" to indicate which entry in the GDT or LDT describes the segment in which the address is located, and an offset value is used to indicate how far above the segment base the particular address is located. When paging is not enabled, the linear address represents the physical address in memory. With paging enabled, the 32-bit linear address is divided into fields representing an index into entries in a page directory 36 (Fig. 1) containing page tables 42, an index into the page table 42 referenced by the index into the directory, and an offset from the beginning of the indicated page to the physical memory location.

An interrupt descriptor table ("IDT") 44 includes entries that are gates corresponding to the handling routines for up to 256 interrupt handlers. Only a single IDT is permitted, and the address of the IDT is stored in an interrupt descriptor table register ("IDTR") 46 (Fig. 3).

The Intel Architecture supports multitasking. Tasks are switched either on the basis of time sharing among tasks or on the occurrence of an event that requires handling by a new task. When switching between tasks, it is necessary to store information about the task that is relinquishing control of the computer system. Upon returning to a task, it is necessary to recall the exact state of the CPU before the task was switched. The state of a task is stored in a segment of memory called a task state segment ("TSS").

Fig. 4 shows a TSS 48 used in a Pentium microprocessor. The TSS includes the values that were stored in many of the CPU registers 12 at the time the task was switched. TSS 48 also includes an LDT selector 50, that is, an index into the GDT, to identify a descriptor in the GDT corresponding to the segment containing an LDT used by the task. The CPU carries out a task switch when it encounters an instruction for the CPU to jump to or call a task gate or load a new TSS 48 into the task register 64. The task gate includes a TSS selector corresponding to the new task. The CPU saves values stored in the CPU registers 12 in a TSS corresponding to the current task and loads into CPU registers 12 the values for the new task. The TSS selector 58, base address 60, and limit 62 of the new TSS are entered into a task register ("TR") 64 (Fig. 3).

Because the above-described Intel Architecture is complex and not designed to concurrently run two operating systems that fully implement protected mode, it has not been possible to combine a fully implemented, protected-mode, real-time operating system with a fully implemented, protected-mode, run-time operating system.

Intel Corporation was able to combine DOS and Windows running in standard mode under DOS with iRMX (Real-time Multitasking eXecutive), a 32-bit, protected-mode, real-time operating system, in a product called "RMX for Windows." DOS, however, operates in "real mode" and does not use the multitasking, protected-mode features of the Intel Architecture. "Real" in the context of "real mode" means that the memory is addressed more directly; "real mode" is unrelated to "real time." Real-mode systems such as DOS do not use descriptor tables. The physical address of a byte of memory is determined by adding the content of a segment register multiplied by 16 to the content of an index register; the sum is a 20-bit address. In real mode, 256 interrupt vectors are stored in the least significant 1024 bytes of the memory space. Each vector is a jump address to a corresponding interrupt handler.

As described in Rajamani et al., BYTE, 17:4, at 119-130 (April 1992), Intel's iRMX for Windows product combines DOS, with or without Windows, with iRMX. The iRMX for Windows product runs DOS as a single task under the iRMX operating system by booting DOS and loading a special iRMX TSR (terminate-stay-resident) program, which reserves a small portion of conventional memory for iRMX. Next, the iRMX for Windows loader loads iRMX into extended memory, where it takes control of the CPU in protected mode and initializes each of the operating system's internal layers and system tasks.

This initialization creates a special system task and dispatches it in virtual 8086 ("V86") mode to the loader's return address and returns to the DOS COMMAND.COM program, which resumes the DOS idle loop. DOS is now set up as an iRMX task in V86 mode, and all subsequent DOS programs will execute in this context. Standard-mode Windows can be started from DOS, if desired, to provide the familiar Windows GUI. Standard-mode Windows will run in the same iRMX task that DOS does. An application's real-time components can then be loaded as protected-mode iRMX jobs.

The iRMX operating system runs in protected mode at protection level 0, while DOS runs in V86 mode. If standard-mode Windows is loaded, it runs under DOS in protected mode but at a less privileged level than iRMX, thereby allowing iRMX to take control of the CPU when required. iRMX controls the GDT and IDT, as well as its own LDT.

Standard-mode Windows obtains extended memory for its own use from iRMX and creates an LDT to manage its memory. The iRMX operating system schedules native iRMX tasks along with the DOS/Windows task according to priority, an attribute of iRMX tasks. iRMX switches the CPU between V86 mode and protected mode as required when switching between tasks or dispatching interrupt handlers.

To ensure that iRMX always has the ability to take over the CPU to respond to a real-time process, the system traps hardware interrupts that DOS/Windows handles in protection level 0 and calls the appropriate real-mode handler or Windows protected-mode handler. iRMX also traps the return from these handlers to ensure that the interrupted task resumes execution in the appropriate CPU mode. Similarly, iRMX traps all software interrupts from DOS/Windows programs and deflects them to the appropriate DOS/Windows handlers. iRMX also traps any attempt by Windows to switch the CPU between real and protected mode. iRMX for Windows performs all mode switching. The iRMX TSR program serves as a surrogate DOS process in whose context iRMX can obtain various DOS services.

When combined with DOS and Windows, iRMX can continue to provide a predictable response time for several reasons. First, because the priority of DOS/Windows tasks is low, higher priority real-time tasks can preempt DOS/Windows tasks at any time. Second, because DOS can run in V86 mode, iRMX can use 386 protection features to trap any attempt to disable CPU interrupts from DOS or Windows. The actual CPU interrupt flag is always set to enable interrupts whenever DOS/Windows is running. However, iRMX maintains the DOS virtual state of this flag so that iRMX will continue to service any hardware interrupts while DOS/Windows has disabled CPU interrupts. This ensures the integrity of DOS/Windows while it provides an upper bound on the latency for iRMX hardware interrupts. Third, iRMX traps DOS/Windows' I/O instructions that modify the interrupt masks of the programmable interrupt controller so that DOS cannot change the mask for iRMX's interrupt levels. Finally, iRMX ensures that tasks above a certain priority level do not take DOS/Windows interrupts. Thus, an application task can guard itself from interruptions, such as a long list of TSRs triggered by the timer.

DOS and Windows in standard mode under DOS do not fully implement protected-mode operations. They do not require their own GDT and IDT. The method used in iRMX for Windows could not be successfully applied to combine a fully protected-mode, run-time operating system, such as Windows NT, with a real-time operating system. The Intel Architecture allows for only a single GDT and IDT, but each fully protected-mode operating system needs a GDT and an IDT to function. If both operating systems shared a single GDT and IDT, the real-time operating system would not be able to preempt the run-time operating system when necessary to service an interrupt within the required amount of time. iRMX could not simply trap requests for lower level services from a fully protected mode operating system as it did in encapsulating DOS.

DOS functions more like an application loader than a protected-mode operating system and can, therefore, run in V86 mode under the control of another operating system. In V86 mode, DOS runs in real mode on a virtual 8086 machine formed by the hardware and a virtual 8086 monitor program. Windows runs as an extension of DOS in V86 mode, it runs essentially as a protection level 3 application. When Windows attempts to establish a GDT and an IDT, it causes a fault because access to the corresponding registers requires a protection level of 0. The fault allows iRMX to intercept those calls and create a virtual IDT for Windows without disturbing the real IDT. iRMX allows Windows to create an LDT and allocates to Windows a segment to manage with its LDT. Because Windows is running at protection level 3, any attempt to access lower protection level interrupts causes a fault, allowing iRMX to intercept the call and control the protection level and priority level of its execution. Thus, Windows running under DOS in V86 mode disables interrupts that would prevent iRMX from responding to real-time applications within the required time limits.

Encapsulating DOS within another operating system, even with standard-mode Windows running as a DOS extension, is relatively easy because the protected-mode mechanism causes faults when DOS attempts to access instructions that could interfere with iRMX real-time functions, and such faults are easily trapped by iRMX. A fully protected operating system, such as Windows NT, runs at a protection level of 0 and, therefore, has the authority to perform all the functions that, under DOS, caused faults and allowed iRMX to intervene. It would not be possible, therefore, for iRMX to detect and deflect a Windows NT command that would cause iRMX to miss a deadline.

Therefore, computer users have been unable to combine the advantages of a fully protected, real-time operating system with the advantages of a fully protected operating system having a strong GUI.

## Summary of the Invention

An advantage of the invention is to be able to encapsulate one fully protected-mode operating system within a second fully protected-mode operating system.

Another advantage of the invention is to provide a real-time operating system with a familiar GUI.

A further advantage of the invention is to provide the advantages of the iRMX real-time operating system with the familiar Windows NT operating system for its user interfaces.

The invention provides a method for encapsuling a fully protected-mode operating system within a second fully protected-mode operating system. The invention is particularly useful for encapsulating a first operating system having a strong GUI capability within a second real-time operating system that can respond to input within a predetermined time to control a process.

In a preferred embodiment, a user starts a computer system and begins using the familiar GUI of the first operating system. The invention provides an interface driver that runs under the first operating system to handle switching between operating systems. The interface driver is part of an operating system encapsulation mechanism ("OSEM"). In response to a user or system request to start the second operating system, the interface driver loads a copy of the boot image of the second operating system into memory. The interface driver saves information about the second operating system and initializes a number of variables that will be shared by both operating systems to coordinate switching between the operating systems. In some embodiments, the linear addresses of the GDT, IDT, and TSS of the second operating system are saved, as are the addresses of the GDT, IDT, and TSS of the first operating system.

Typically, the addresses of certain parts of the second protected-mode, real-time operating system are loaded into the appropriate CPU registers, and the current task becomes the second operating system, at which time the second operating system runs its initialization code and initializes all its subsystems. The second operating system, which is now controlling the computer, runs an encapsulation subsystem, which is also a part of the OSEM. The encapsulation subsystem prepares a native task to run the first protected-mode operating system.

In one embodiment, the encapsulation subsystem causes a hardware task switch back to the first operating system, which continues running the interface driver task that was interrupted by the switch to the second operating system. The interface driver reloads into the appropriate registers the saved addresses of certain parts of the first operating system. In some embodiments, the interface driver loads the addresses of the GDT and IDT used by the first operating system. The task switch causes the LDT, page directory address, and other register content saved in the TSS to be reloaded. In other embodiments, the actions to switch operating systems that would normally be performed by the hardware during a task switch are performed by individual instructions in the switching software.

With the first operating system running as a task under the second real-time operating system, the interface driver manages the events that cause a switch back to the second operating system. Such events include a clock tick that the second operating system clock handler needs to service, a hardware interrupt that a second operating system interrupt handler needs to service, and communications between a first operating system application, service provider, process, or thread and a second operating system task.

To insure real-time behavior in the second operating system, no events under the second operating system will explicitly cause a switch to the first operating system. However, the second operating system will monitor clock ticks and will notify the interface driver in the first operating system of missed first operating system clock ticks so the timers of the first operating system can be adjusted to reflect the time that passed while the second operating system was running.

By allowing the second, real-time operating system to encapsulate the first operating system, debuggers native to both environments can be run simultaneously, thereby allowing concurrent debugging of the interface and the real-time portion of a system.

The only part of the first operating system that is aware of the second operating system is the interface driver. This driver provides all the interfacing to the second operating system. By loading the location of parts of the second operating system before switching, the first operating system can be unaware that the second operating system exists and has underlying control of the computer system. The first operating system is operating in its normal environment. The second operating system can interrupt any task of the first operating system to ensure that the second operating system can respond within its predetermined time to external or internal input.

## Brief Description of the Drawings

Fig. 1 shows conceptually some of the registers and memory in a typical microprocessor used as the CPU of the computer system shown in Fig. 5.

Fig. 2 shows the structure of a GDT.

Fig. 3 shows the memory management registers of an Intel Pentium microprocessor that can be used to implement the present invention.

Fig. 4 shows the structure of a TSS.

Fig. 5 shows a typical computer system in which the present invention is implemented.

Fig. 6 is a flow chart showing the steps used to start the INtime operating system from the Windows NT operating system.

Figs. 7a-7e are block diagrams showing the structural changes that occur while performing the steps shown in Fig. 6.

Fig. 8 is a flow chart showing the steps used to return control of the computer system to the Windows NT operating system that is running as a task in the INtime operating system.

Fig. 9 shows the structure of the messaging system used to send messages between tasks running under the two operating systems.

Figs. 10a and 10b are flow charts showing the servicing of a real-time extension request (NTX) and illustrating the use of the messaging system of Fig. 9.

Fig. 11 shows the steps in a second preferred embodiment for switching from Windows NT operating system to INtime operating system.

Fig. 12 shows the status of various tables and registers during the process of changing between operating systems.

Fig. 13 shows the steps in the second preferred embodiment for switching back from INtime to Windows NT.

Fig. 14 shows the structure of the messaging system used to send messages between tasks running under the two operating systems.

Detailed Description of Preferred Embodiment

In a preferred embodiment, the invention allows the encapsulation of the Windows NT operating system by Microsoft Corporation, Redmond, Washington, within the INtime real-time operating system by RadiSys Corporation, Hillsboro, Oregon, the assignee of the present invention. The INtime operating system is based on the iRMX operating system originally developed by Intel Corporation.

In naming programs such as applications and drivers, this specification first lists the operating system under which the program or driver is running. For example, a Windows NT INtime driver runs under Windows NT and includes instruction code related to the INtime operating system.

Fig. 5 shows a computer system 66 in which the present invention is implemented. Computer system 66 includes devices, such as keyboard 68 and mouse 70, by which a user can input information into computer system 66, and includes a video display terminal 72, such as a cathode ray tube or a liquid crystal display, by which a user can perceive output from computer system 66. Computer system 66 is based upon the Intel Architecture and uses a Pentium or Pentium Pro or later microprocessor as its CPU 10. Tables, registers, and memory organization for CPU 10 function as described above. Computer system 66 also includes one or more mass storage devices, such as a floppy disk drive 76 or a hard disk drive 78.

Fig. 6 is a flow chart showing the steps of encapsulating the Windows NT operating system within the INtime operating system. Fig. 7 conceptually illustrates the structure resulting from the steps shown in Fig. 6.

Fig. 6 shows in step 84 that the Windows NT operating system is first loaded into computer system 66 in a normal manner. In step 86, Windows NT loads its normal device drivers (not shown) and a Windows NT INtime driver 88 running at protection level 0. Windows NT INtime driver 88 is the portion of the OSEM that runs under the Windows NT operating system. Windows NT INtime driver 88 obtains from Windows NT a block of contiguous, non-paged memory into which the INtime kernel is later loaded. Fig. 7a shows that Windows NT and Windows NT INtime driver 88 are loaded in computer system 66. The underlining of

Windows NT in the figure indicates that it is running and in control of computer system 66. The dotted line surrounding Windows NT INtime driver 88 in Fig. 7a indicates that the running of Windows NT INtime driver 88 is controlled by the surrounding Windows NT operating system.

Step 90 and Fig. 7b show a user or another program initiates the running of a Windows NT INtime loader application 94, which runs at protection level 3. In an alternative embodiment, Windows NT INtime driver 88 is not loaded with the other NT drivers upon the boot-up of Windows NT, but is loaded by Windows NT INtime loader application 94 upon its activation. This alternative is less desirable because the large amount of contiguous, non-pageable memory required to load the INtime kernel is more readily available at start-up.

Windows NT INtime loader application 94 obtains from Windows NT INtime driver 88 the address of the block of memory and loads into it a boot image 96 (Fig. 7b) of the INtime operating system (step 98). Boot image 96 is essentially the complete INtime operating system as it would be loaded if INtime were used to start or boot up, computer system 66. The INtime boot image includes an INtime GDT, IDT, and TSS. Windows NT INtime loader application 94 puts information into the INtime TSS that will cause the INtime operating system to execute its initialization code the first time that NT switches to the INtime hardware task. The boot image is changed from the standard INtime operating system to recognize that INtime has been loaded by Windows NT, which has already performed certain initialization functions, such as programming the programmable interrupt controllers and the programmable interrupt timer. INtime, then, is not required to perform those initializations that have already occurred.

Step 100 shows that Windows NT INtime driver 88 saves in global variables the linear addresses of the loaded INtime GDT, IDT, and TSS and initializes a number of variables that will be shared by both INtime and itself. These variables include the addresses of the two TSSes representing the hardware tasks of the running Windows NT operating system and the INtime operating system being loaded, the linear addresses of the Windows NT GDT and IDT, and the selector, that is, the index of the descriptor, of the Windows NT TSS in the Windows NT GDT. Selected descriptors are created in the INtime GDT that correspond to the protection level 0 and protection level 3 code and data segments that are in the Windows NT GDT, and another descriptor is created that corresponds to the Windows NT TSS. Therefore, portions of the Windows NT INtime driver 88 can continue to function when the GDTR 32 and the IDTR 46 are switched to point to the INtime GDT and IDT in step 102.

In step 102, Windows NT INtime driver 88 loads the addresses of INtime GDT and IDT, along with a null LDT, into the appropriate CPU registers. A null LDT is required because the INtime GDT does not contain descriptors corresponding to an arbitrary Windows NT LDT that might currently be active. Loading a null LDT selector prevents a possible processor fault when loading the GDT.

After the GDT and the IDT registers are loaded with pointers to the INtime GDT and IDT, respectively, and the addresses of the Windows NT GDT and LDT required to switch back to Windows NT are loaded into global variables, Windows NT INtime driver 88 causes a hardware task switch (step 104) to the INtime operating system. The hardware task switch causes the contents of the CPU registers to be saved in a Windows NT TSS. The saved Windows NT TSS points to the instruction in the Windows NT INtime driver 88 immediately following the instruction that caused the hardware task switch so that upon switching back to the Windows NT task, the system always knows where execution will begin. The hardware task switch also causes the registers of CPU 10 to be loaded with the information stored in the INtime TSS. The hardware task switch also loads a new page directory base in the CR3 control register of CPU 10, so the INtime environment remains isolated from the Windows NT application environment.

The INtime TSS contains the entry point to the INtime initialization code. Step 106 shows that INtime begins initialization immediately upon the task switch, Initialization includes saving the Windows NT CR0 and PIC masks and loading the INtime CR0 and PIC masks. After the task switch, the addresses of the Windows NT LDTs, GDT, and IDT are still stored in memory in global variables, so that their locations can be reloaded into the applicable CPU registers upon switching control of computer system 66 back to Windows NT.

Fig. 7c shows INtime running computer system 66. The INtime operating system initialization code brings up various INtime subsystems, such as the system debugger, the shared C-Library, and the Paging Subsystem, which provides virtual memory management under the Windows NT operating system.

Step 108 shows that the initialization of the INtime operating system includes running an INtime Windows NT encapsulation subsystem 114 (Fig. 7c), which is the INtime portion of the OSEM. INtime Windows NT encapsulation subsystem 114 prepares an INtime task, the INtime Windows NT task 116, to run Windows NT under the INtime operating system. Termination block 118 shows that the switch to INtime is complete, and INtime is now controlling the operation of computer system 66.

Fig. 8 is a flow chart showing the steps used to return control of the computer system 66 to the Windows NT operating system running as a task in the INtime operating system. Step 120 shows that the process of returning control of computer system 66 to the encapsulated Windows NT operating system is initiated either when an INtime task requests a service for a Windows NT service or when all INtime tasks are complete or waiting, thus allowing the lower priority INtime Windows NT task to run.

Step 122 shows that INtime Windows NT encapsulation subsystem 114 saves the INtime CR0 and PIC mask and restores the Windows NT CR0 and PIC mask. In step 124, INtime Windows NT encapsulation subsystem 114 performs a hardware task switch back to the Windows NT TSS. The hardware task switch causes the current state of CPU 10 under INtime to be stored in the INtime TSS and the values of the registers that were stored in the Windows NT TSS are reloaded into the appropriate CPU 10 registers (step 126). The address of the LDT of the INtime Windows NT task 116 is automatically reloaded from the TSS into the LDTR as part of the hardware task switch. The hardware task switch also reloads the page directory base address into the CR3 register, so Windows NT can access the memory it was using. Upon switching back to Windows NT, the Windows NT INtime driver 88 begins to execute at the instruction immediately following the one that caused the task switch to INtime.

The Windows NT INtime driver 88 includes a Windows NT INtime interface exchange event procedure 128 that handles the task switching. Windows NT INtime interface exchange event procedure 128 loads the Windows NT GDT and IDT (step 132). Terminal block 134 shows Windows NT running while fully encapsulated within INtime. Fig. 7d shows Windows NT running and controlling computer system 66, unaware that it is running as a task within the INtime operating system. Windows NT INtime driver 88 is the only part of the Windows NT operating system that is aware INtime is running.

When Windows NT is running as a task under INtime, the Windows NT INtime driver 88 manages the events that cause a switch back to the INtime operating system. Such events include a clock tick that the INtime clock handler needs to service, a hardware interrupt that an INtime interrupt handler needs to service, and communications between a Windows NT application, service provider, process, or thread and an INtime task. Fig. 7e shows computer system 66 back under control of INtime.

The INtime Windows NT encapsulation subsystem 114 manages the various events that cause a switch back to the Windows NT environment. Such events include communication between an INtime task and a Windows NT application, service provider, process, or thread; and returning control to the Windows NT environment when the Windows NT task becomes the running INtime task.

An INtime task can be in a number of action states. For example, a task can be running, ready, delayed, or asleep. INtime tasks also have priority attributes. The Windows NT encapsulation INtime task is set up as the lowest priority INtime task, so it becomes the default idle task. When other INtime tasks have stopped running because they are delayed, that is, waiting for an event, INtime Windows NT task 116 will run and will totally consume the CPU with administrative functions, such as refreshing screens. If one or more INtime tasks become ready, based on a clock tick or an interrupt, then the higher priority ready INtime tasks will preempt the low-priority INtime Windows NT task 116. The INtime tasks will run until they have completed the current event processing, at which time they will no longer be ready. Then, the only, and, therefore, the highest priority, ready task remaining will be the INtime Windows NT task 116, which will become the running task.

Thus, computer system 66 will typically spend the majority of its time running the idle Windows NT encapsulation INtime task. However, all INtime real-time tasks have higher priority than the Windows NT task and can preempt it as needed. Thus, computer system 66 will switch to Windows NT either when an INtime task requires a Windows NT service or when no other INtime task is ready.

As described above, task switching from Windows NT to INtime is performed by the OSEM, which includes the Windows NT INtime interface exchange event procedure 128 on the Windows NT side and the INtime Windows NT encapsulation subsystem 114 on the INtime side. Thus, all switching performed between the two operating systems involves a single interface program in each operating system, and each operating system will begin executing instructions at a known point in a predetermined program. This centralized switching interface makes possible a messaging mechanism between operating systems, and, therefore, the creation of real-time extensions of the Windows NT operating system.

Fig. 9 shows the structures associated with the messaging system. The INtime scheduler 148, driven by a task going to sleep, will cause a switch to Windows NT through the INtime Windows NT encapsulation subsystem 114. When Windows NT passes control back to INtime, the INtime Windows NT encapsulation subsystem 114 will run the instruction immediately after the one that caused the previous switch.

Windows NT may initiate a switch to INtime when, for example, a Windows NT interrupt needs to be serviced by INtime, shown in block 150, or when a Windows NT application makes a real-time extension (NTX) call to INtime, shown in block 152. Before Windows NT INtime interface exchange event procedure 128 switches to the INtime TSS, Windows NT INtime interface exchange event procedure 128 writes into the INtime TSS information about the reason for the switch. Windows NT INtime interface exchange event procedure 128 writes a reason class into the EAX register 154 of the INtime TSS. Windows NT INtime interface exchange event procedure 128 writes a reason code into the EBX register 156 of the INtime TSS, and Windows NT INtime interface exchange event procedure 128 writes a 32-bit reason data pointer into the EDI register 158 of the INtime. The 32-bit reason data pointer is not needed in all operating system switches and typically points to a memory structure having data that the incoming operating system will require to carry out the reason for the switch. Immediately after the hardware task switch to INtime, INtime Windows NT encapsulation subsystem 114 begins to run the instruction immediately following the instruction that caused the previous switch to Windows NT. INtime Windows NT encapsulation subsystem 114 will determine from the codes in the INtime TSS why INtime was called and will call the appropriate INtime procedure to process the request.

For example, if INtime is called from Windows NT to service an interrupt, the class code will indicate that the switch is caused by an interrupt request, and the reason code will indicate which interrupt is requested. INtime Windows NT encapsulation subsystem 114 then dispatches a call to the corresponding INtime interrupt handler. Similarly, if the class and reason codes indicate that the switch is to transfer a messaging request, INtime Windows NT encapsulation subsystem 114 dispatches the task to the INtime message handler.

Similarly, just before control is switched to Windows NT, INtime Windows NT encapsulation subsystem 114 will write message codes in the Windows NT TSS. Windows NT INtime interface exchange event procedure 128 will begin to run after the switch at the instruction immediately following the instruction that caused the switch to INtime. Windows NT INtime interface exchange event procedure 128 can then interpret the reason for the switch. For example, the reason may be to hand off of a message pointer or simply that no INtime tasks are ready, so control is being yielded to the default task, Windows NT.

Table 1 contains a list of class codes used in the messaging operation. Table 2 shows a list of reason codes used in the messaging operation. Skilled persons will understand that additional class and reason codes can be created and used for different applications.

| Table 1 | |
|---|---|
| **Reason Classes/Codes for RMX/NT Task Switch Coordination** | |
| RMXIF_RMX_INTERRUPT_CLASS | 10000H |
| Task switch because of hardware (HW) interrupt for Windows NT. | |
| | |
| RMXIF_NT_INTERRUPT_CLASS (Not used) | 20000H |
| | |
| RMXIF_RMX_SERVICE_REQUEST_CLASS | 30000H |
| Task switch because of request for Windows NT services from INtime (includes responses). | |
| | |
| RMXIF_NT_SERVICE_REQUEST_CLASS | 40000H |
| The semantics of this class may evolve differently from the generic service request class. | |
| | |
| RMXIF_RTE_CLASS | 50000H |
| Task switch reason codes. | |
| These are passed to the target task in EBX. | |
| | |
| REASONS FOR RMXIF_RMX_INTERRUPT_CLASS (NT → RMX) | |
| The reason code for this class is the RMX IDT index for the interrupt being forwarded from NT. Codes range from 48 to 63 decimal. | |
| | |
| Reasons for RMXIF_RMX_SERVICE_REQUEST_CLASS (NT → RMX) | |
| First service request of RMX. Initializes the OS. | |
| | |
| RMXIF_RMX_SERVICE_INITIALIZE_RMX | 0001H |
| NT requests RMX to shut down. This is called when the RMXIF driver unloads, which would happen during an NT shutdown. | |

| Table 1 | |
|---|---|
| **Reason Classes/Codes for RMX/NT Task Switch Coordination** | |
| | |
| RMXIF_RMX_SERVICE_SHUTDOWN_RMX | 0002H |
| No specific service requested of RMX. | |
| Use this to yield the CPU back to NT. | |
| | |
| RMXIF_RMX_SERVICE_YIELD_TO_RMX | 0003H |
| | |
| Reasons for RMXIF_NT_SERVICE_REQUEST_CLASS (RMX → NT) | |
| RMX uses this to signal NT that it has shut down. | |
| Do not switch back to RMX after this event. | |
| After an RMX shutdown, RMX HW task should always reply with this. | |
| | |
| RMXIF_NT_SERVICE_STOP_RMX | 0001H |
| RMXIF_NT_SERVICE_YIELD _TO_NT | 0002H |

| Table 2 | |
|---|---|
| **NTX Call Reason Codes** The number of reason codes and supported NTX calls may evolve from this initial listing. | |
| RMXIF_RTE_GET_TASK_TOKENS | 01H |
| RMXIF_RTE_LOOKUP_OBJECT | 02H |
| RMXIF_RTE_SEND_UNITS | 03H |
| RMXIF_RTE_RECEIVE_UNITS | 04H |
| RMXIF_RTE_SEND_DATA | 05H |
| RMXIF_RTE_RECEIVE_DATA | 06H |
| RMXIF_RTE_SEND_MESSAGE | 07H |
| RMXIF_RTE_RECEIVE_MESSAGE | 08H |

| | |
|---|---|
| RMXIF_RTE_GET_TYPE | 09H |
| RMXIF_RTE_GET_SIZE | 0AH |
| RMXIF_RTE_GET_ADDRESS | 0BH |
| LAST_RTE_FUNCTION | 0BH |

The present invention makes it possible to implement real-time extensions to the Windows NT operating system in INtime. (NTX APIs) An NTX call allows a Windows NT thread to communicate with an INtime task using an INtime exchange object. Figs. 10a and 10b show the steps involved in making an NTX call from Windows NT.

Step 166 of Fig. 10a shows a Windows NT user application requesting an NTX service. For example, assume that an INtime task has created a mailbox and is waiting there. The INtime task has catalogued the name of this mailbox in the root job of INtime. A Windows NT application that needs to find the mailbox can make an NTX call from the Windows NT operating system to an NTX provider in INtime. In step 168, the Windows NT application accesses NTX functions through a Windows NT INtime dynamic link library ("DLL"), which in turn sends the appropriate I/O control codes to the Windows NT INtime driver 88. The dispatch procedure of the Windows NT INtime driver 88 recognizes the received I/O control code as an NTX request and calls Windows NT INtime interface exchange event procedure 128 to pass the NTX request to INtime. The Windows NT INtime interface exchange event procedure 128 writes the appropriate class code, reason codes, and data pointer into the INtime TSS as described above in step 170 and, in step 172, causes the hardware task switch to INtime.

In this example, the class code 50000H in the EAX identifies the class of the request as an NTX request. The reason code in the EBX memory location of the INtime TSS indicates which real-time extension request is being passed. In this example, the 02H code in the EBX memory location indicates that the request is LOOK UP OBJECT. In step 174, INtime Windows NT encapsulation subsystem 114 interprets the information in the INtime TSS memory fields, and in step 176, it determines whether the incoming request requires a blocking function. If the NTX request does not require a blocking function, step 178 shows that the request is handed off to an appropriate INtime procedure that, using the data pointer stored in the EDI memory location of the INtime TSS, can obtain the rest of the information it needs to carry out the NTX request. After the INtime procedure finds the INtime mailbox (step 182), the INtime procedure returns to the INtime Windows NT encapsulation subsystem 114, which, in step 188, writes the class and reason code, as well as a pointer to the mailbox token, into the Windows NT TSS and in step 190 causes a hardware task switch back to Windows NT. In step 192, Windows NT INtime interface exchange event procedure 128 interprets the reason for the return to Windows NT and sends the returned information to the thread that requested it.

The same Windows NT task that requested the lookup could then, for example, send a segment to that mailbox in the same way. The value of the class code written into the INtime TSS EAX register for the new request would also be 50000H because the new request is also a request for a real-time extension service. The code 06H in the EBX memory location would be different from that of the previous request because the NTX service being requested is different. Likewise, the data pointer in the EDI memory location would be different.

A request for a real-time extension service may or may not include a blocking function, that is, a function that can block if the request cannot be completed immediately. Because Windows NT runs as a single task under INtime, blocking a Windows NT request would completely block Windows NT. To allow other threads to continue running under Windows NT, step 194 of Fig. 9b shows that a request with a blocking function is handed off to a surrogate INtime task. INtime has a pool of tasks for carrying out various NTX requests. Decision block 196 shows that if the INtime task cannot proceed because it is waiting for an event, it will enter a sleep state 202, and other INtime tasks can run. If other INtime tasks are ready to run (step 204), INtime will run them (step 206). If other INtime tasks are ready to run, INtime Windows NT encapsulation subsystem 114 will yield control back to Windows NT (step 208).

When control switches back to Windows NT and the reason code does not indicate that the NTX request is complete, Windows NT understands that the request has not been serviced and treats the request as a pending interrupt, in the same manner as Windows NT would treat, for example, waiting for data to be available from a SCSI disk drive (step 210). One skilled in the art would be familiar with the Windows NT system for blocking requests.

When the event that the surrogate INtime task had been waiting for (step 212) occurs, the surrogate INtime task wakes up, completes the NTX request, post its results to the "completed NTX" mailbox, sets a flag, and returns to a sleep state to be available to service other requests. The next time INtime is ready to yield back to Windows NT, INtime Windows NT encapsulation subsystem 114 checks the flag (step 222) and determines that there is a completed real-time extension request to return to Windows NT. INtime Windows NT encapsulation subsystem 114 will retrieve the completed request from the "completed NTX" mailbox, and then write the class and reason codes into the EAX and EBX memory locations and the data pointer in the EDI memory location of the Windows NT TSS (step 224) and complete the task switch to Windows NT (step 226). When Windows NT encapsulation application 94 begins to run, Windows NT INtime driver 88 recognizes that it has a completed NTX request and will unblock the calling thread and return the results of the request to it (step 228). Thus, the present invention solves the problem of communicating across operating systems and provides real-time extensions for the run-time operating system.

To ensure that INtime maintains control of its interrupts, it is desirable to make certain extensions to the hardware abstraction layer (the "HAL") of Windows NT to prevent it from modifying the Windows NT IDT without INtime being informed. Windows NT has a HAL; INtime does not. A HAL allows an operating system, such as Windows NT, to be more easily ported to different hardware platforms by concentrating the features that vary between platforms into one layer. INtime, on the other hand, has historically been closely associated with the Intel architecture.

The HAL services hardware, such as the interrupt structure used in IBM-compatible personal computers. The interrupt structure uses two programmable interrupt controllers ("PICs") -- a master PIC and a slave PIC tied into master level 2. The PICs assign hardware interrupt levels to various pieces of hardware, such as disk drives. The hardware interrupt levels are then mapped to software handlers through the IDT. Although Windows NT does not have access to the INtime IDT, Windows NT can change the programming of the PICs, which could change interrupt handling by INtime, as well as by Windows NT. For example, Windows NT could mask, that is, disable, an interrupt through the PIC. Also, because Windows NT allows drivers to be loaded while it is running, Windows NT could enable and assign new hardware-interrupt levels. Windows NT masking an interrupt could stop required input to INtime from a real-time task. If Windows NT enables a new hardware-interrupt level, INtime needs to be able to handle the interrupt appropriately. In a preferred embodiment, the HAL is extended so that INtime is informed when Windows NT attempts to mask an interrupt or when a new driver tries to register. INtime can then handle the change appropriately on the INtime side, or modify or deny these attempted changes to the PIC by Windows NT.

The HAL also controls the clock rates. The HAL is preferably modified so that if Windows NT attempt to change the clock rate, INtime will be informed because all of INtime's timings depend on the clock rate.

Lastly, the Intel Architecture in protected mode has the ability to detect hardware faults, such as attempting to read a page of memory that has not been loaded, corrupted stacks, and other similar situations. Those hardware faults are dealt with in the HAL, and serious errors would cause Windows NT to cease functioning. The HAL is preferably modified so that if Windows NT is disabled, INtime has an opportunity to respond, if possible, and continue to run its real-time tasks.

Another difficulty in combining fully protected-mode operating systems has been that it would be impossible to use debuggers for both systems concurrently. Debuggers use the first sixteen entries in the IDT. For example, interrupt 1, the first entry in the IDT, causes computer system 66 to "single step," that is, to stop after execution of each source code instruction so that a programmer can determine the result of executing each line of code. Interrupt 3, the third entry in the IDT, causes a break from the program being debugged to a monitor program that monitors the program execution. The monitor program required to monitor the functioning of an INtime program is different from the monitor program required to monitor a Windows NT program. Because fully protected-mode operating systems allow only a single IDT, it has been impossible to run debuggers for both systems concurrently because the IDT entries are unsuitable for one or the other debugger.

The present invention, which stores and reloads the IDTs when switching between operating systems, allows Windows NT environment debuggers, such as SoftICE from NuMega Technologies and CodeView from Microsoft Corporation, and INtime environment debuggers, such as SDM/SDB, which is provided with the INtime operating system, and Soft-Scope from Concurrent Science Inc., to run simultaneously, thereby allowing concurrent debugging of the Windows NT and the real-time portions of a system.

By loading the INtime GDT and IDT before doing the hardware task switch to the INtime TSS, normal Windows NT operation does not need to know anything about the presence of the INtime operating system. All the segments, INtime interrupt handlers, and the INtime GDT, IDT, and TSS are only defined in the INtime GDT. The Windows NT GDT has no corresponding descriptors and thus is not perturbed in any way as part of the encapsulation process.

INtime, on the other hand, maintains the segments used by Windows NT in its GDT and so controls the operation of Windows NT as it does other tasks. This control by INtime allows it to maintain the predictability required of a true real-time operating system while encapsulating a 32-bit, protected-mode operating system having strong GUI capabilities.

Figs. 11, 12, 13, and 14 illustrate a second preferred embodiment for encapsulating Windows NT within the INtime operating system. As in the first embodiment, an Operating System Encapsulation Mechanism, or OSEM 230, administers the encapsulation and the switching between operating systems. The OSEM in the second embodiment differs from the one in the first embodiment as described below. The second embodiment is more preferred than the first.

In the second embodiment, OSEM 230 instructions save and load information that, in the first embodiment, is saved and loaded by the hardware during execution of a task switch. Because the switch between operating systems is performed manually, registers that are unchanged during a switch are not unnecessarily and automatically saved and reloaded. Also, because the EAX, EBX, and EDI registers are not automatically changed, the messaging code information in those registers is more readily accessible to the program. Details not specifically described below with respect to the second embodiment are the same as those described with respect to the first embodiment.

A portion of the OSEM runs under each of the two operating systems. Fig. 11 shows the sequence of event for encapsulating Windows NT under INtime in the second embodiment. Terminal block 238 shows that, as in the previous embodiment, computer system 66 boots up into the first operating system, Windows NT. At boot-up, Windows NT loads its regular drivers and a Windows NT interface driver 240, which is part of the OSEM 230 running under Windows NT. Windows NT interface driver 240 allocates memory into which INtime Kernel 266 will be loaded. Although the large block of memory required for INtime Kernel 266 is more readily available at start-up, in some embodiments, it may be possible to load Windows NT interface driver 240 after start-up.

Fig. 12a shows Windows NT normally running an application program 242 and having an associated GDT 244, 272, IDT 274, TSS 276, PDIR 278, and PIC masks 280 loaded. Step 246 shows that an event invokes a kernel loader 268, which is configured as an Windows NT service and can be invoked either automatically at start up after Windows NT interface driver 240 is loaded or after start-up by request. Kernel loader 268 loads the INtime Kernel 266 boot image into the memory that was previously allocated for the boot image by Windows NT interface driver 240. (Step 248). Automatic invocation of Kernel loader 268 at start-up is preferred in an end-user application, such as process controller, whereas a developer may prefer to invoke kernel loader 268 at his convenience. Kernel loader 268 also fills in an INtime TSS to cause the INtime operating system to execute its initialization code the first time that Windows NT switches to the INtime hardware task.

In step 250, Windows NT interface driver loads a GDT 258 and an IDT 260 for the second operating system. Fig. 12b shows the status of computer 66 after step 250. In step 262, interface driver 240 jumps to INtime Kernel OSEM portion 264 under the INtime Kernel 266. OSEM portion 264 saves the Windows NT PDIR 278 in a global variable in step 286 and loads the INtime Kernel PDIR 288 in step 290. In step 292, the INtime Kernel OSEM portion 264 then loads the INtime Hardware TSS 294 into TR 64. The INtime Kernel OSEM portion 264 saves in global memory the Windows NT PIC mask and the contents of the Windows NT control register CR0 in step 300 and updates the contents of CR0 in step 302 to be consistent with the operation of the INtime Kernel 266. The INtime Kernel OSEM portion 264 next loads the INtime PIC masks 304 in step 306. The status of computer system 66 is now shown in Fig. 12c. Now, the INtime operating system is fully loaded and ready to run an INtime task, as shown in terminal block 308. The INtime Kernel OSEM portion 264 determines the appropriate task to run based upon the contents of the EAX, EBX, and EDI registers, as described above with respect to the first embodiment. Fig. 12d shows computer system 66 running an INtime task under the INtime operating system.

Fig. 13 shows the steps for returning control of computer 66 from the INtime operating system to the Windows NT operating system. When all activity running under the INtime operating system is complete, that is, all tasks are either completed or waiting for an event, as shown in terminal block 310, the INtime task that encapsulates all of Windows NT runs again and initiates in step 316 the switch back to Windows NT. In step 318, INtime Kernel OSEM portion 264 restores the Windows NT floating point context that was saved if an INtime task used the numeric processor, thereby changing the context from the one that was in use when Windows NT last controlled computer 66. In step 320, INtime Kernel OSEM portion 264 then restores the Windows NT updated PIC masks 280, CR0 322, and PDIR 278. Step 328 shows that INtime Kernel OSEM portion 264 updates the EAX, EBX, and EDI registers to indicate the return status to the Windows NT portion of the OSEM interface driver 240. In step 330, the INtime Kernel OSEM portion 264 loads a pointer to the Windows NT TSS into the TR. The status of computer system 66 after step 330 is shown in Fig. 12e. In step 332, the INtime Kernel OSEM portion 264 starts running the Windows NT interface driver 240. Fig. 12f shows computer system after step 332. In step 334, the Windows NT interface driver 240 loads the Windows NT GDT 244, 272 and IDT 274. Windows NT is now fully loaded and running with the appropriate GDT, IDT, TSS, PDIR, and PIC masks. Step 336 shows that Windows NT returns to the code and thread it was running when the environment switching event occurred. Lastly, terminal block 340 shows that Windows NT continues to control computer 66 until an event causes a return to the INtime operating system.

Fig. 14 shows the messaging system in the second embodiment. Fig. 14 is similar to Fig. 9, but as shown in Fig. 14, the reason class, reason code, and reason data pointer are in the EAX, EBX, and EDI register, and not in the register data of the incoming TSS.

Many changes may be made to the above-described details of the preferred embodiment of the present invention without departing from the underlying principles thereof. The scope of the present invention should, therefore, be determined only by the following claims.

## Claims

1. A method of encapsulating a first fully protected mode operating system within a second fully protected mode operating system on an Intel architecture-based computer system including a microprocessor that executes a current task and that contains memory management registers, comprising:

loading and running the first operating system;

providing an encapsulating program within the first operating system running at a privileged protection level, the encapsulating program performing the following steps;

loading into memory a copy of at least a part of the second operating system;

storing the values used by the first operating system to manage memory and contained in the memory management registers;

storing the value used by the first operating system to track the state of its tasks;

loading into memory management registers values corresponding to parts of the second operating system;

switching the current hardware task of the microprocessor to the initialization program of the second operating system;

running the initialization program of the second operating system; and

running an encapsulation subsystem to create a task corresponding to the first operating system to run under the second operating system, thereby enabling switching control of the computer system back to the first operating system operating as a task under the second operating system

no part of the first operating system, other than the encapsulation program, having any memory management information corresponding to memory used by the second operating system so that the first operating system runs normally without being aware that it is encapsulated within the second operating system.

2. The method of claims 1 in which the first operating system is Windows NT and the second operating system is INTIME or iRMX.

3. The method of claim 2 in which the microprocessor is a member of the Intel x86 family of microprocessors values and the values used by the microprocessor to manage memory include the location of a global descriptor table and an interrupt descriptor table.

4. The method of claim 3 in which storing the value used by the first operating system to track the state of its tasks includes storing at least one task state segment selector.

5. The method of claim 2 in which switching the current hardware task includes storing in a task state segment a selector corresponding to a local descriptor table for the current task and the base address of the page directory for the current task.

6. A method of switching control from a first, encapsulated protected mode operating system to a second, encapsulating protected mode operating system, comprising:

encapsulating a first operating system within a second operating system in accordance with claim 1;

switching, by the encapsulation subsystem running under the second operating system, the current task of the microprocessor to the encapsulation program within the first operating system; and

loading into the memory management registers the values previously stored.

7. A computer implemented system for providing a familiar graphical user interface and capability of responding in a deterministic manner in real-time, comprising:

a user input device;

a display terminal for providing information to the user;

a microprocessor;

a main memory;

a first fully protected mode operating system stored in the main memory, the first operating system program capable of providing a graphical user interface to the user by responding to user input through the user input device and displaying information to the user on the display terminal; and

a second fully protected mode operating system, the second fully protected operating mode system being a real-time operating system capable of responding within a predetermined time to external, asynchronous events, the microprocessor being able to concurrently provide a user the familiar graphical user interface of the first fully protected mode operating system and deterministic response of the second fully protected mode operating system.

8. A method of encapsulating a first fully protected mode operating system within a second fully protected mode operating system on an Intel architecture-based computer system including a microprocessor that executes a current task and that contains memory management registers, comprising:

loading and running the first operating system;

loading an interface driver that operates under the first operating system;

allocating memory for a boot image of the second operating system;

loading a boot image of the second operating system;

loading information required by the second operating system to begin running;

running the second operating system; and

saving information required by the first operating system and loading additional information required by the second operating system corresponding to the information saved, the second operating system encapsulating the first so that the first operating system cannot prevent the second operating system from responding in a real-time, deterministic manner to an event.

9. The method of claim 8, further comprising a system for passing messaging between the operating systems.

10. The method of claim 9 in which the messaging system stores information in segment registers.

11. The method of claim 8 in which loading information required by the second operating system to begin running includes loading a global descriptor table and an interrupt descriptor table.

12. The method of claim 8 in which saving information required by the first operating system includes saving a page directory, the contents of a control register, and a programmable interrupt controller mask.

13. The method of claim 8 further comprising switching back to the first operating system by:

updating segment registers to indicate the status upon return;

saving information relating to the operating of the second operating system and loading information related to the first operating system;

continuing to run a thread that was running before the switch to the first operating system.

FIG. 1

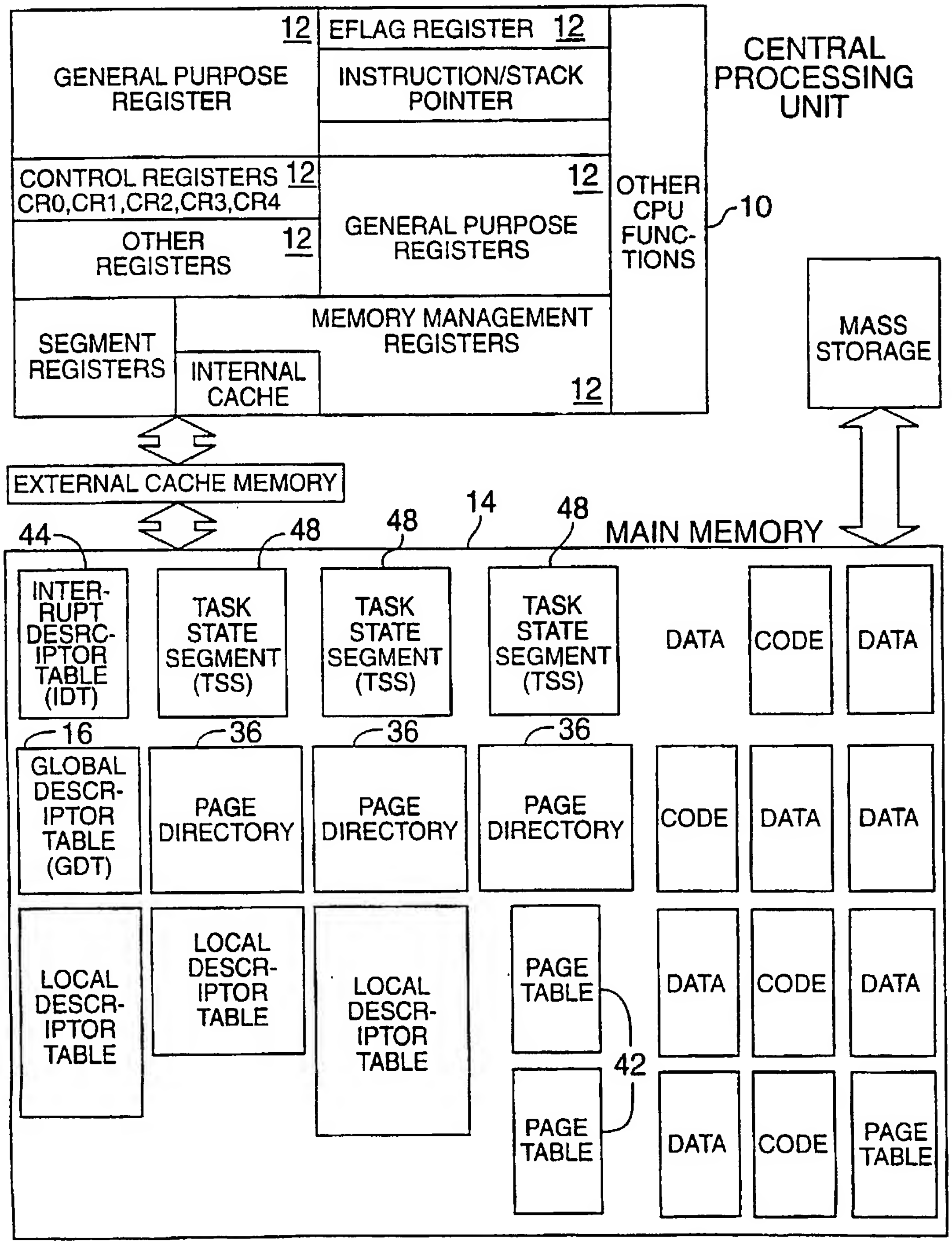
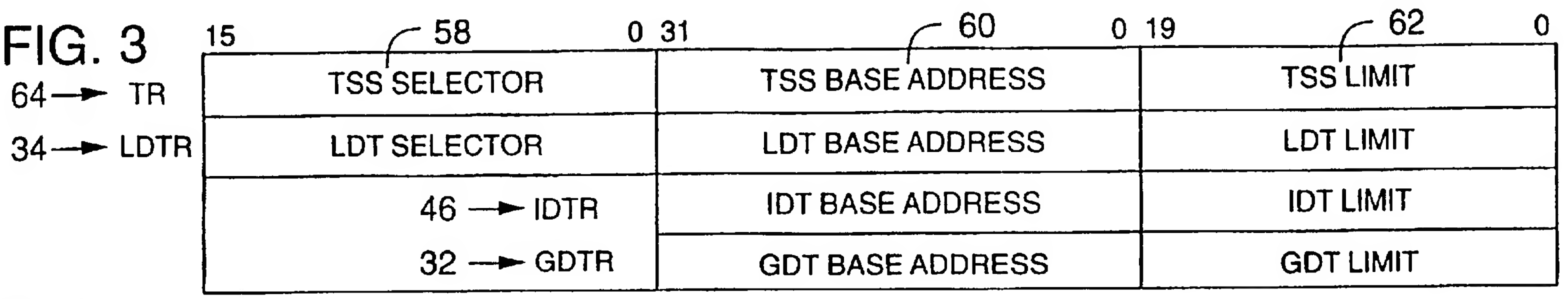
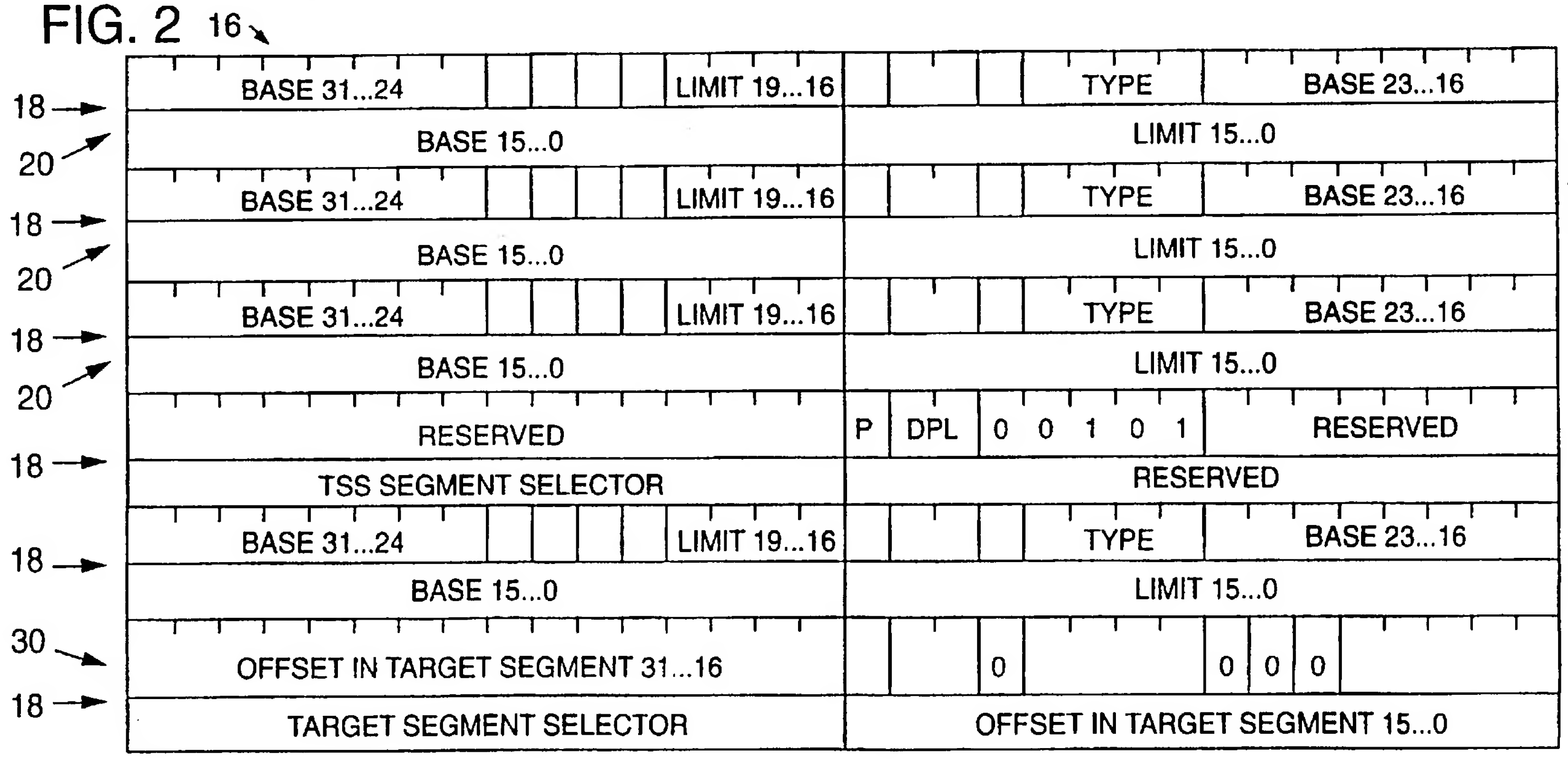

GENERAL PURPOSE REGISTER 12
EFLAG REGISTER 12
INSTRUCTION/STACK POINTER
CONTROL REGISTERS CR0,CR1,CR2,CR3,CR4 12
OTHER REGISTERS 12
GENERAL PURPOSE REGISTERS 12
OTHER CPU FUNCTIONS
CENTRAL PROCESSING UNIT
10
SEGMENT REGISTERS
INTERNAL CACHE
MEMORY MANAGEMENT REGISTERS 12
MASS STORAGE
EXTERNAL CACHE MEMORY
44
48
48
14
48
MAIN MEMORY
INTERRUPT DESCRIPTOR TABLE (IDT)
TASK STATE SEGMENT (TSS)
TASK STATE SEGMENT (TSS)
TASK STATE SEGMENT (TSS)
DATA
CODE
DATA
16
36
36
36
GLOBAL DESCRIPTOR TABLE (GDT)
PAGE DIRECTORY
PAGE DIRECTORY
PAGE DIRECTORY
CODE
DATA
DATA
LOCAL DESCRIPTOR TABLE
LOCAL DESCRIPTOR TABLE
LOCAL DESCRIPTOR TABLE
PAGE TABLE
42
PAGE TABLE
DATA
CODE
DATA
DATA
CODE
PAGE TABLE

FIG. 3



64 TR
34 LDTR
46 IDTR
32 GDTR
15
58
0
31
60
0
19
62
0
TSS SELECTOR
TSS BASE ADDRESS
TSS LIMIT
LDT SELECTOR
LDT BASE ADDRESS
LDT LIMIT
IDT BASE ADDRESS
IDT LIMIT
GDT BASE ADDRESS
GDT LIMIT

FIG. 2



16
18
20
18
20
18
20
18
18
30
18
BASE 31...24
LIMIT 19...16
TYPE
BASE 23...16
BASE 15...0
LIMIT 15...0
BASE 31...24
LIMIT 19...16
TYPE
BASE 23...16
BASE 15...0
LIMIT 15...0
BASE 31...24
LIMIT 19...16
TYPE
BASE 23...16
BASE 15...0
LIMIT 15...0
RESERVED
P
DPL
0 0 1 0 1
RESERVED
TSS SEGMENT SELECTOR
RESERVED
BASE 31...24
LIMIT 19...16
TYPE
BASE 23...16
BASE 15...0
LIMIT 15...0
OFFSET IN TARGET SEGMENT 31...16
0
0 0 0
TARGET SEGMENT SELECTOR
OFFSET IN TARGET SEGMENT 15...0

FIG. 4

50

| 31 16 | 15 0 |
|---|---|
| I/O MAP BASE | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 T |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | TASK LDT SELECTOR |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | GS SELECTOR |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | FS SELECTOR |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | DS SELECTOR |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | SS SELECTOR |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | CS SELECTOR |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ES SELECTOR |
| EDI | |
| ESI | |
| EBP | |
| ESP | |
| EBX | |
| EDX | |
| ECX | |
| EAX | |
| EFLAG | |
| EIP | |
| CR3 (PDBR) | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | SS FOR CPL2 |
| ESP FOR CPL2 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | SS FOR CPL1 |
| ESP FOR CPL1 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | SS FOR CPL0 |
| ESP FOR CPL0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | BACK LINK TO PREVIOUS TSS |

48

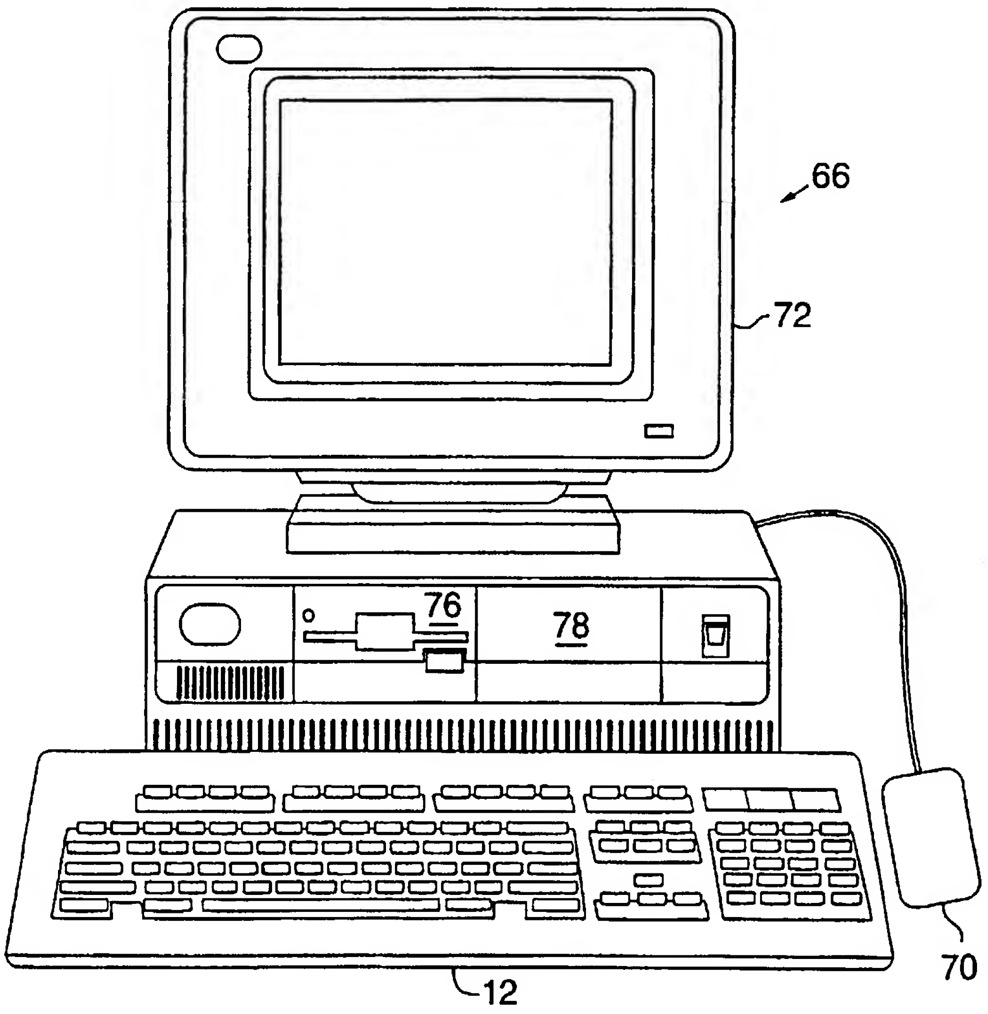


FIG. 5

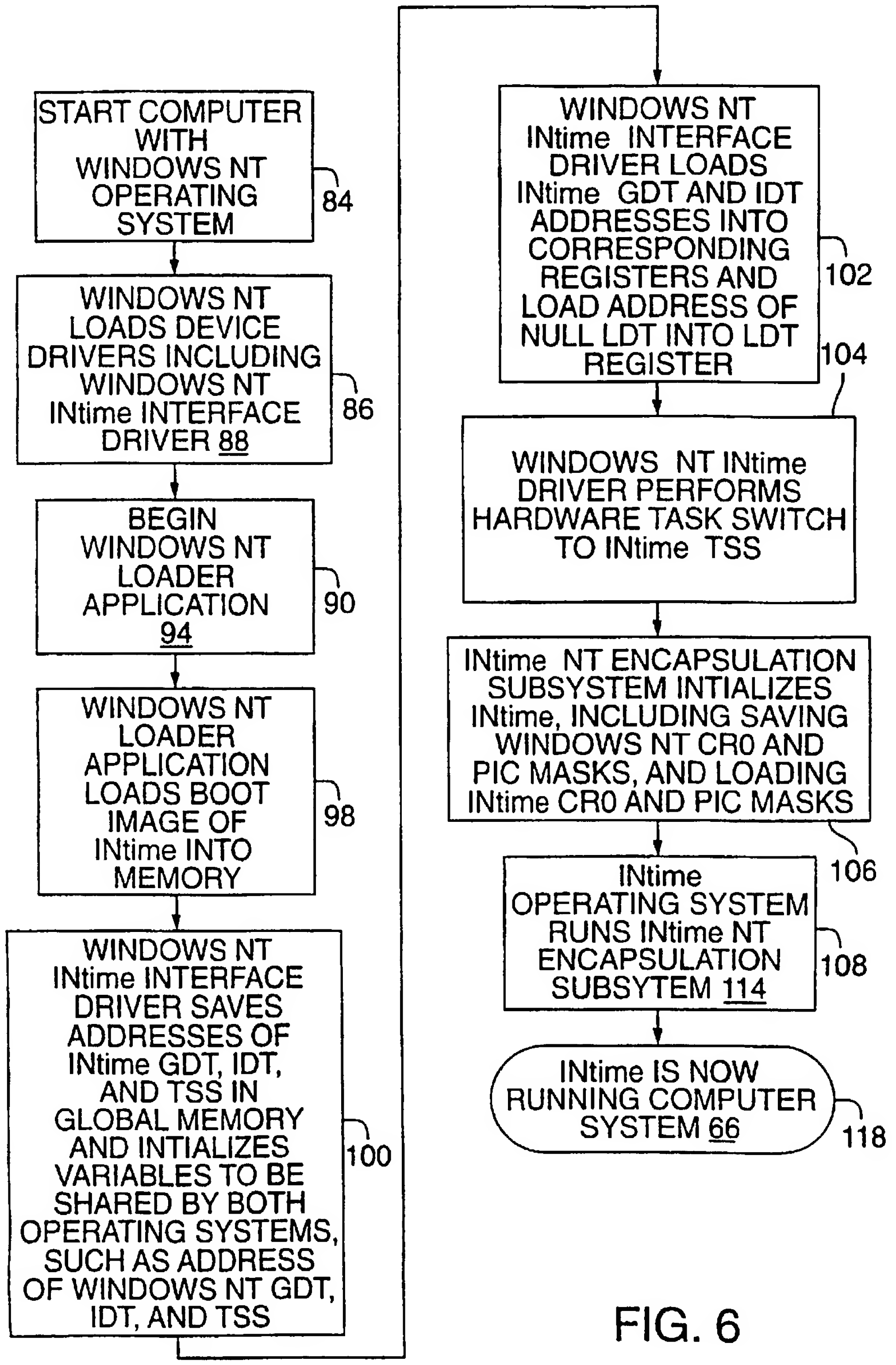

START COMPUTER WITH WINDOWS NT OPERATING SYSTEM
84
WINDOWS NT LOADS DEVICE DRIVERS INCLUDING WINDOWS NT INtime INTERFACE DRIVER 88
86
BEGIN WINDOWS NT LOADER APPLICATION 94
90
WINDOWS NT LOADER APPLICATION LOADS BOOT IMAGE OF INtime INTO MEMORY
98
WINDOWS NT INtime INTERFACE DRIVER SAVES ADDRESSES OF INtime GDT, IDT, AND TSS IN GLOBAL MEMORY AND INTIALIZES VARIABLES TO BE SHARED BY BOTH OPERATING SYSTEMS, SUCH AS ADDRESS OF WINDOWS NT GDT, IDT, AND TSS
100
WINDOWS NT INtime INTERFACE DRIVER LOADS INtime GDT AND IDT ADDRESSES INTO CORRESPONDING REGISTERS AND LOAD ADDRESS OF NULL LDT INTO LDT REGISTER
102
104
WINDOWS NT INtime DRIVER PERFORMS HARDWARE TASK SWITCH TO INtime TSS
INtime NT ENCAPSULATION SUBSYSTEM INTIALIZES INtime, INCLUDING SAVING WINDOWS NT CR0 AND PIC MASKS, AND LOADING INtime CR0 AND PIC MASKS
106
INtime OPERATING SYSTEM RUNS INtime NT ENCAPSULATION SUBSYTEM 114
108
INtime IS NOW RUNNING COMPUTER SYSTEM 66
118

FIG. 6

FIG. 7a

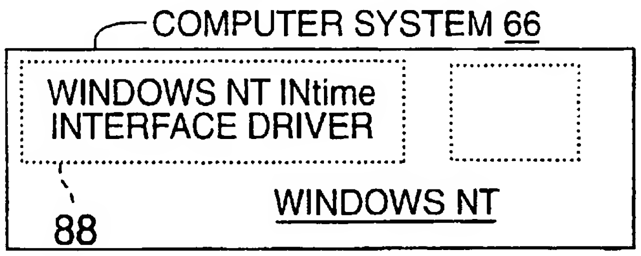


FIG. 7b

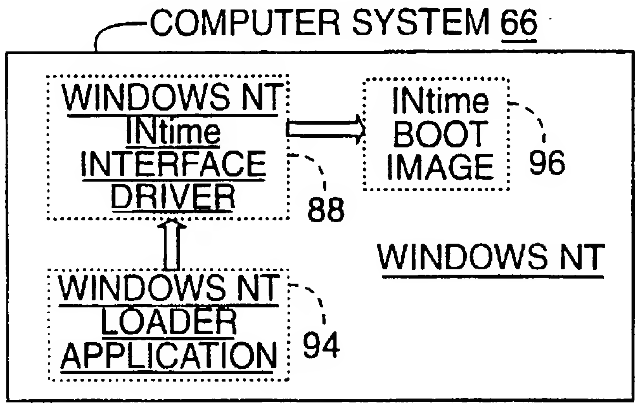

COMPUTER SYSTEM 66
WINDOWS NT INtime INTERFACE DRIVER
88
INtime BOOT IMAGE
96
WINDOWS NT LOADER APPLICATION
94
WINDOWS NT

FIG. 7c

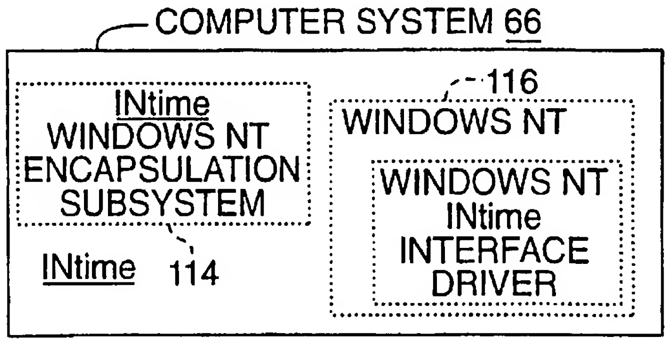

COMPUTER SYSTEM 66
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM
INtime
114
116
WINDOWS NT
WINDOWS NT INtime INTERFACE DRIVER

FIG. 7d

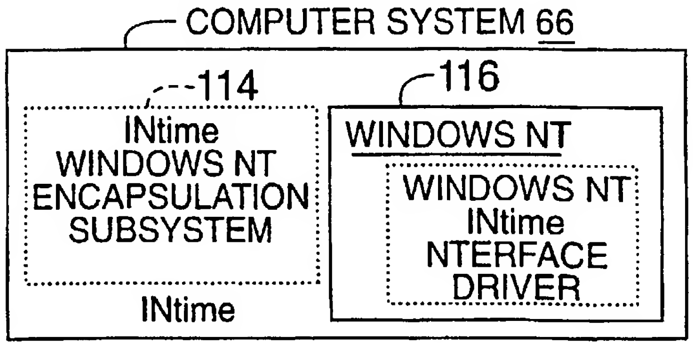

COMPUTER SYSTEM 66
114
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM
INtime
116
WINDOWS NT
WINDOWS NT INtime NTERFACE DRIVER

FIG. 7e

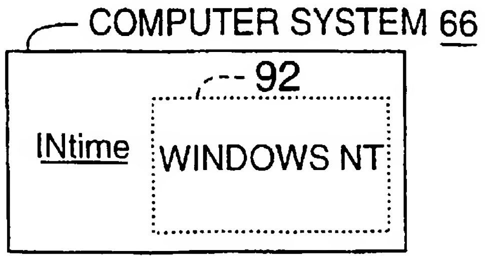


FIG. 8

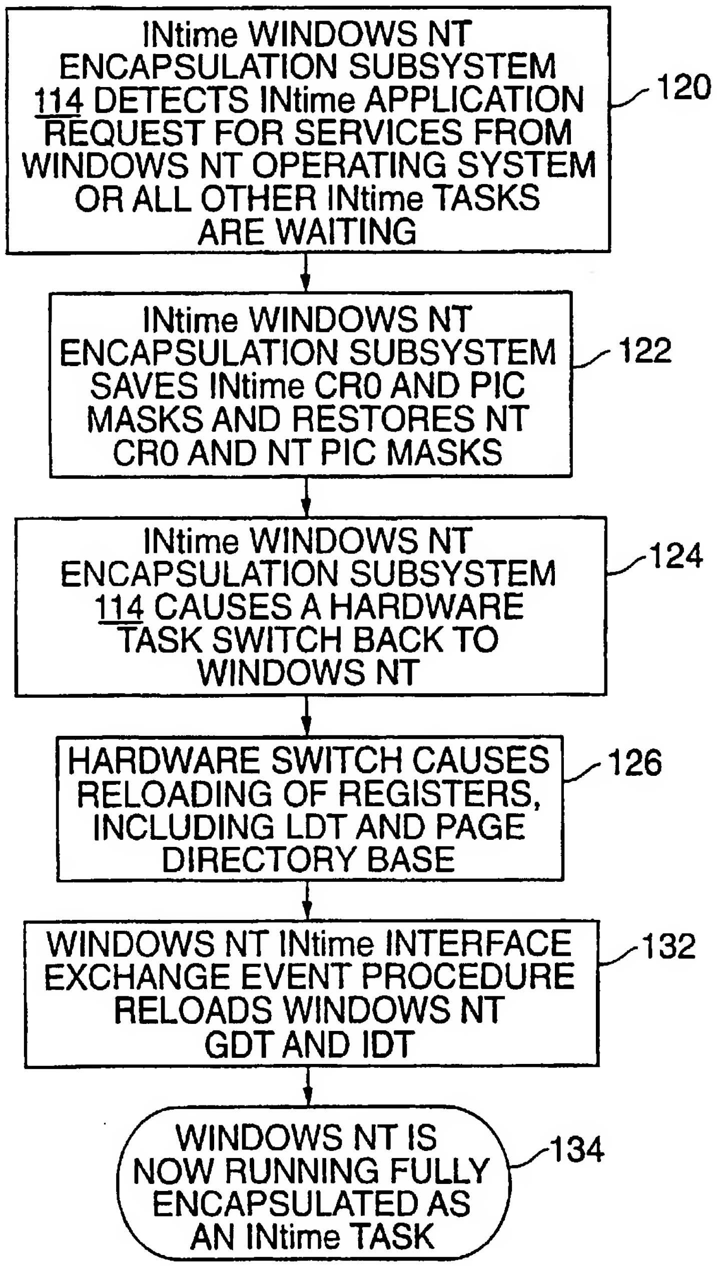

INtime WINDOWS NT ENCAPSULATION SUBSYSTEM 114 DETECTS INtime APPLICATION REQUEST FOR SERVICES FROM WINDOWS NT OPERATING SYSTEM OR ALL OTHER INtime TASKS ARE WAITING
120
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM SAVES INtime CR0 AND PIC MASKS AND RESTORES NT CR0 AND NT PIC MASKS
122
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM 114 CAUSES A HARDWARE TASK SWITCH BACK TO WINDOWS NT
124
HARDWARE SWITCH CAUSES RELOADING OF REGISTERS, INCLUDING LDT AND PAGE DIRECTORY BASE
126
WINDOWS NT INtime INTERFACE EXCHANGE EVENT PROCEDURE RELOADS WINDOWS NT GDT AND IDT
132
WINDOWS NT IS NOW RUNNING FULLY ENCAPSULATED AS AN INtime TASK
134

FIG. 9

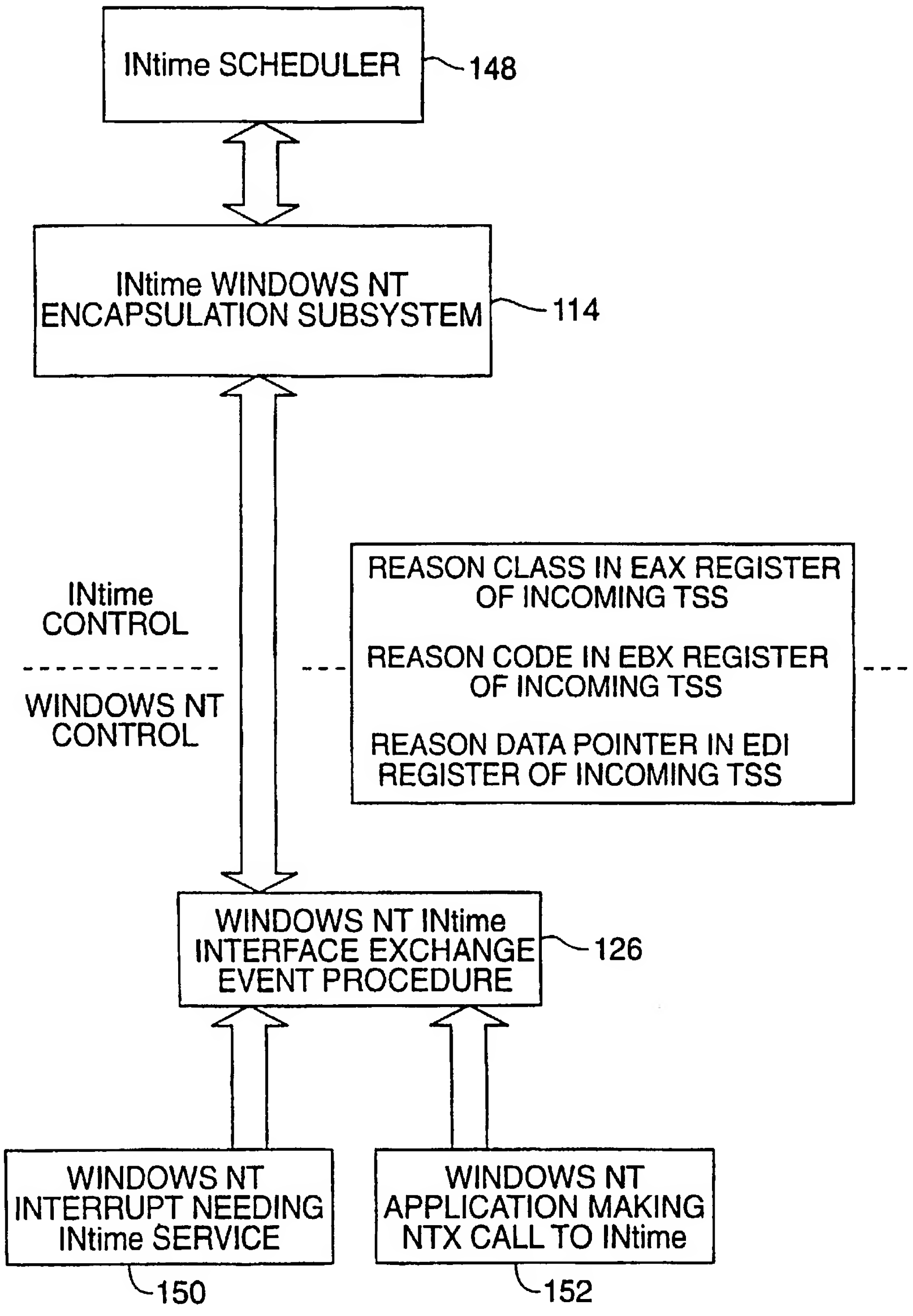
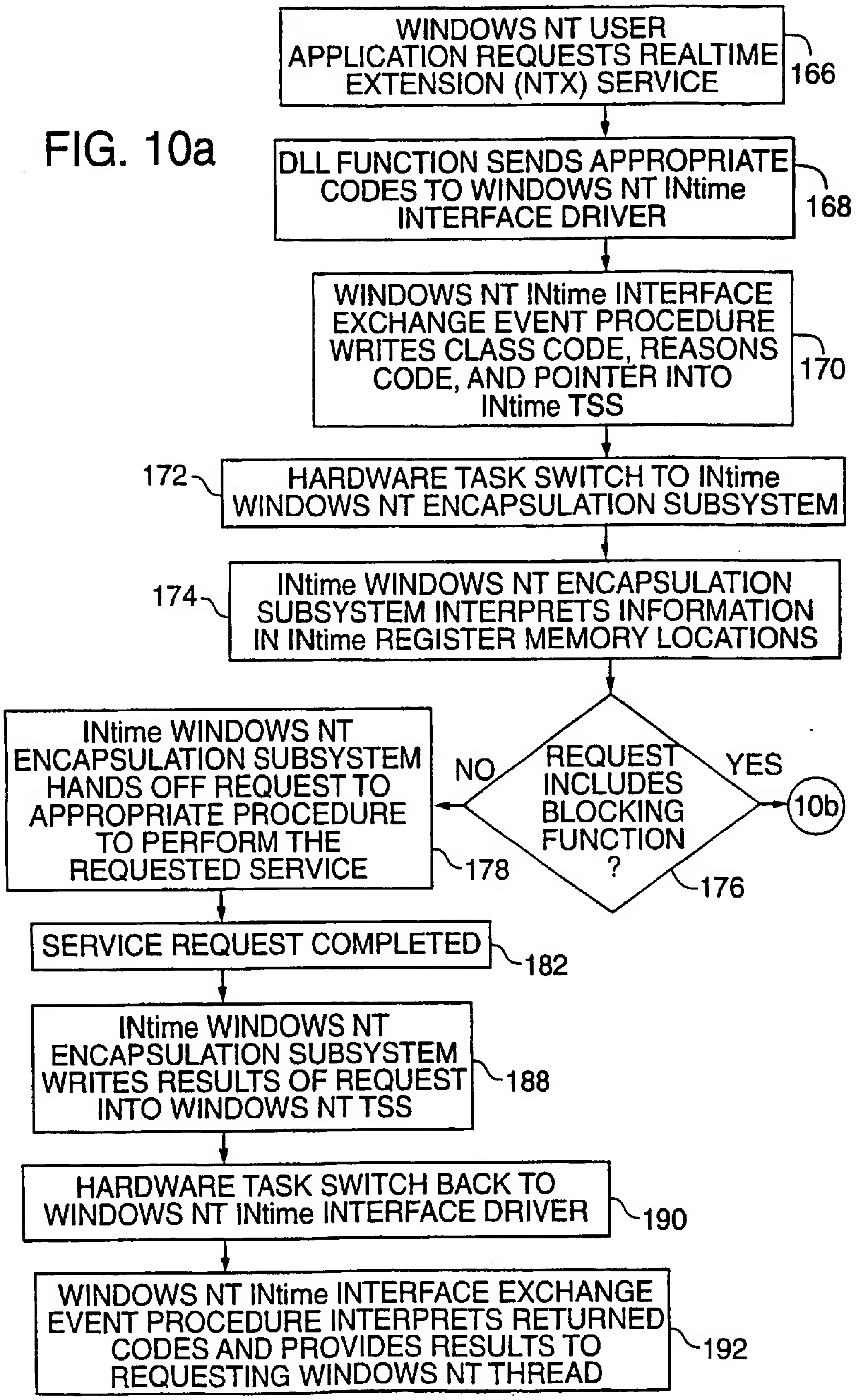

INtime SCHEDULER
148
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM
114
INtime CONTROL
WINDOWS NT CONTROL
REASON CLASS IN EAX REGISTER OF INCOMING TSS
REASON CODE IN EBX REGISTER OF INCOMING TSS
REASON DATA POINTER IN EDI REGISTER OF INCOMING TSS
WINDOWS NT INtime INTERFACE EXCHANGE EVENT PROCEDURE
126
WINDOWS NT INTERRUPT NEEDING INtime SERVICE
150
WINDOWS NT APPLICATION MAKING NTX CALL TO INtime
152

FIG. 10a



WINDOWS NT USER APPLICATION REQUESTS REALTIME EXTENSION (NTX) SERVICE
166
DLL FUNCTION SENDS APPROPRIATE CODES TO WINDOWS NT INtime INTERFACE DRIVER
168
WINDOWS NT INtime INTERFACE EXCHANGE EVENT PROCEDURE WRITES CLASS CODE, REASONS CODE, AND POINTER INTO INtime TSS
170
172
HARDWARE TASK SWITCH TO INtime WINDOWS NT ENCAPSULATION SUBSYSTEM
174
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM INTERPRETS INFORMATION IN INtime REGISTER MEMORY LOCATIONS
REQUEST INCLUDES BLOCKING FUNCTION ?
176
YES
10b
NO
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM HANDS OFF REQUEST TO APPROPRIATE PROCEDURE TO PERFORM THE REQUESTED SERVICE
178
SERVICE REQUEST COMPLETED
182
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM WRITES RESULTS OF REQUEST INTO WINDOWS NT TSS
188
HARDWARE TASK SWITCH BACK TO WINDOWS NT INtime INTERFACE DRIVER
190
WINDOWS NT INtime INTERFACE EXCHANGE EVENT PROCEDURE INTERPRETS RETURNED CODES AND PROVIDES RESULTS TO REQUESTING WINDOWS NT THREAD
192

FIG. 10b

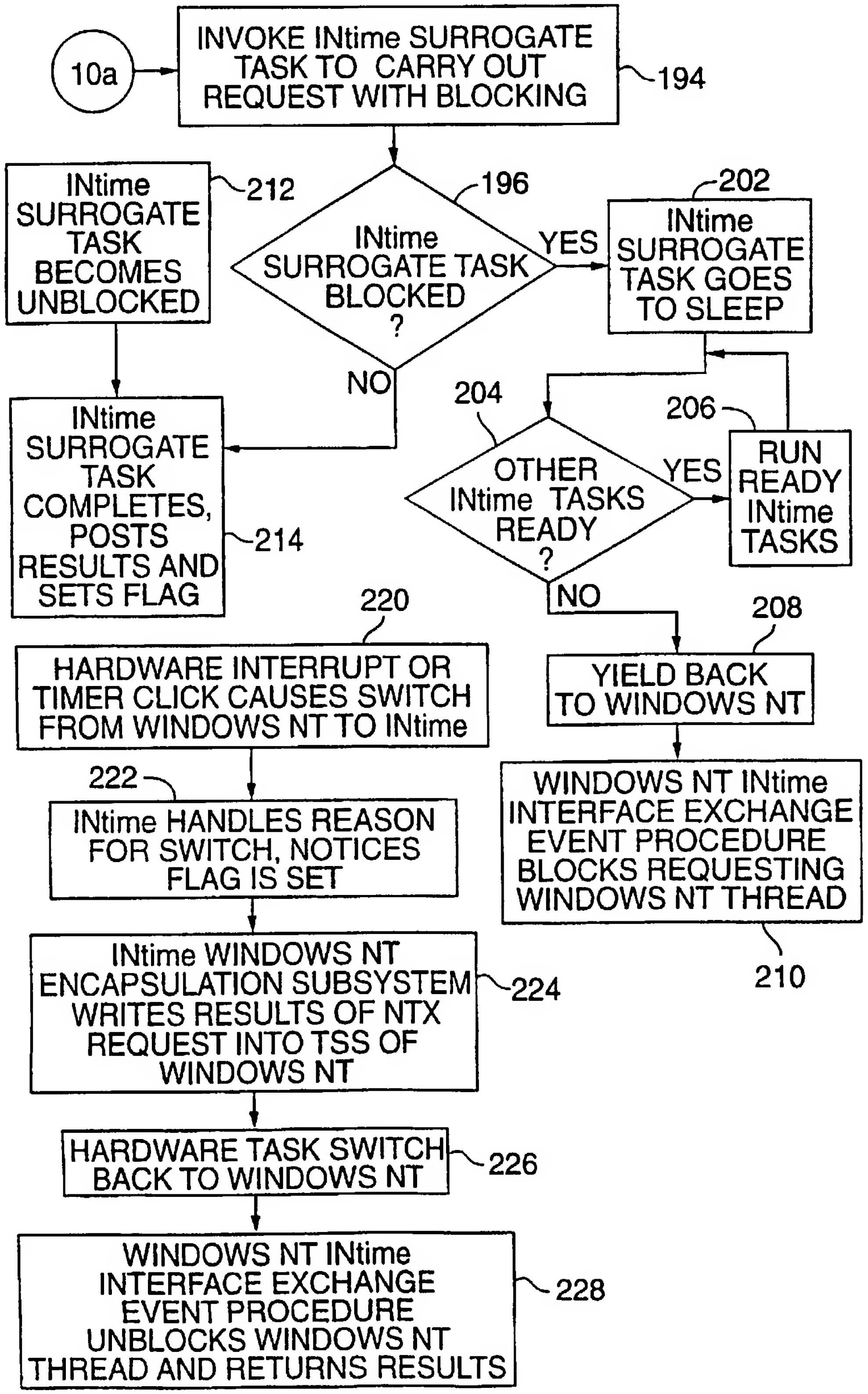

10a
INVOKE INtime SURROGATE TASK TO CARRY OUT REQUEST WITH BLOCKING
194
INtime SURROGATE TASK BLOCKED ?
196
YES
NO
INtime SURROGATE TASK GOES TO SLEEP
202
INtime SURROGATE TASK BECOMES UNBLOCKED
212
INtime SURROGATE TASK COMPLETES, POSTS RESULTS AND SETS FLAG
214
OTHER INtime TASKS READY ?
204
YES
NO
RUN READY INtime TASKS
206
YIELD BACK TO WINDOWS NT
208
WINDOWS NT INtime INTERFACE EXCHANGE EVENT PROCEDURE BLOCKS REQUESTING WINDOWS NT THREAD
210
HARDWARE INTERRUPT OR TIMER CLICK CAUSES SWITCH FROM WINDOWS NT TO INtime
220
INtime HANDLES REASON FOR SWITCH, NOTICES FLAG IS SET
222
INtime WINDOWS NT ENCAPSULATION SUBSYSTEM WRITES RESULTS OF NTX REQUEST INTO TSS OF WINDOWS NT
224
HARDWARE TASK SWITCH BACK TO WINDOWS NT
226
WINDOWS NT INtime INTERFACE EXCHANGE EVENT PROCEDURE UNBLOCKS WINDOWS NT THREAD AND RETURNS RESULTS
228

FIG. 11

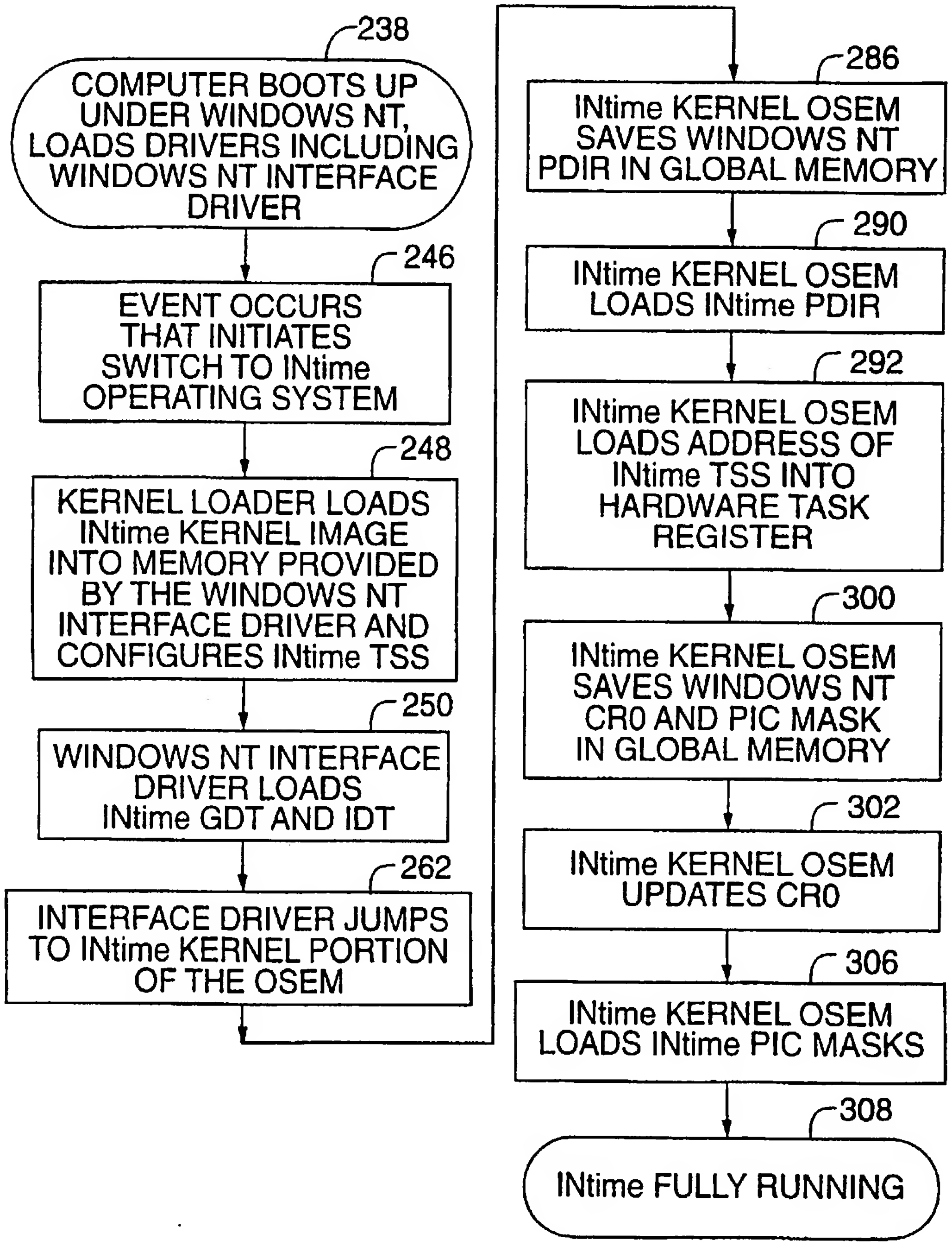

238
COMPUTER BOOTS UP UNDER WINDOWS NT, LOADS DRIVERS INCLUDING WINDOWS NT INTERFACE DRIVER
246
EVENT OCCURS THAT INITIATES SWITCH TO INtime OPERATING SYSTEM
248
KERNEL LOADER LOADS INtime KERNEL IMAGE INTO MEMORY PROVIDED BY THE WINDOWS NT INTERFACE DRIVER AND CONFIGURES INtime TSS
250
WINDOWS NT INTERFACE DRIVER LOADS INtime GDT AND IDT
262
INTERFACE DRIVER JUMPS TO INtime KERNEL PORTION OF THE OSEM
286
INtime KERNEL OSEM SAVES WINDOWS NT PDIR IN GLOBAL MEMORY
290
INtime KERNEL OSEM LOADS INtime PDIR
292
INtime KERNEL OSEM LOADS ADDRESS OF INtime TSS INTO HARDWARE TASK REGISTER
300
INtime KERNEL OSEM SAVES WINDOWS NT CR0 AND PIC MASK IN GLOBAL MEMORY
302
INtime KERNEL OSEM UPDATES CR0
306
INtime KERNEL OSEM LOADS INtime PIC MASKS
308
INtime FULLY RUNNING

FIG. 12a

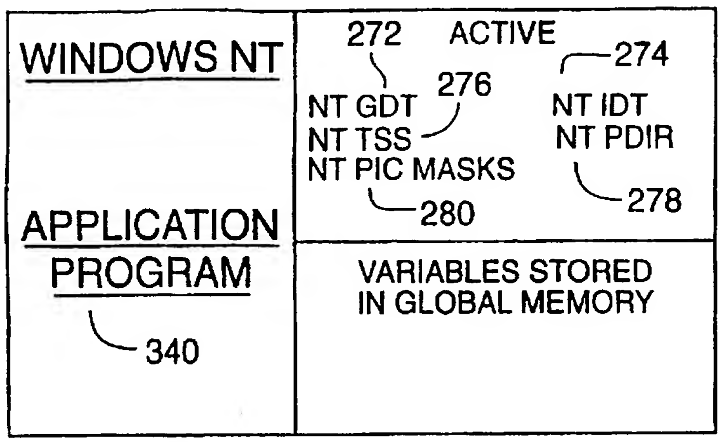


FIG. 12b

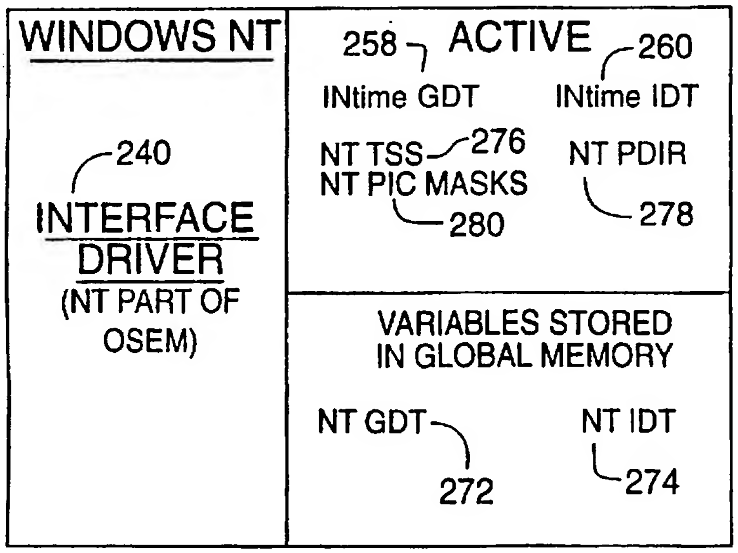


FIG. 12c

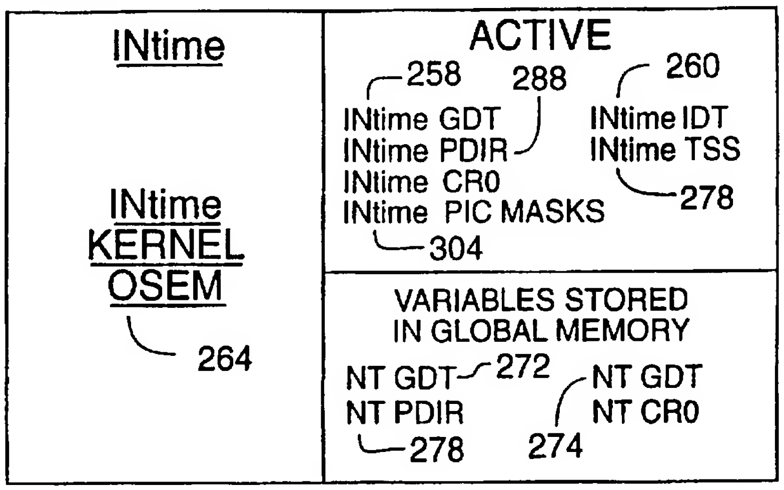
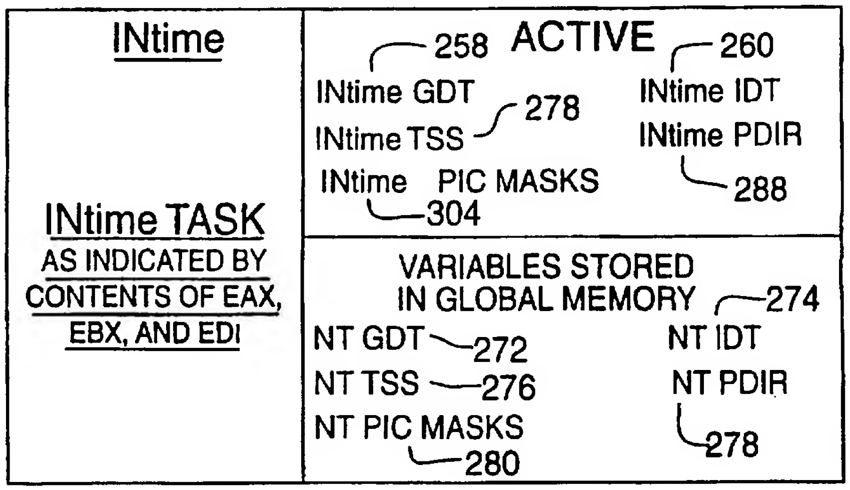


FIG. 12d



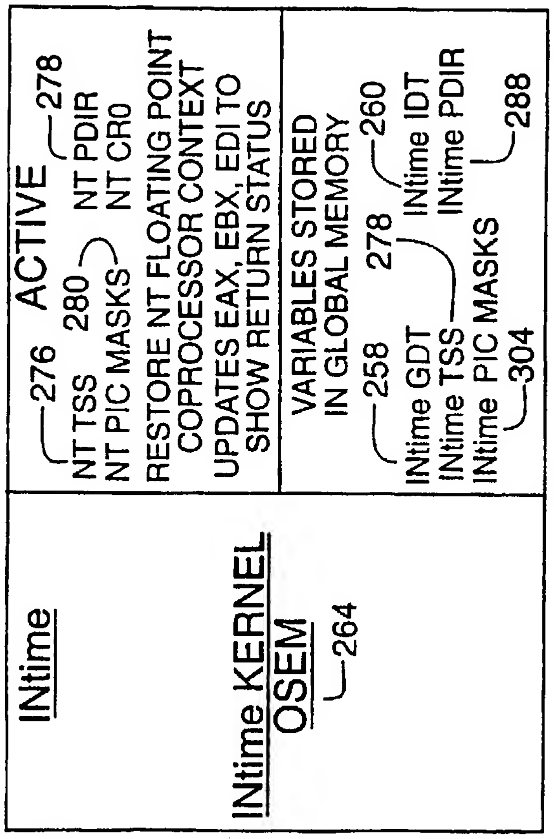


FIG. 12e

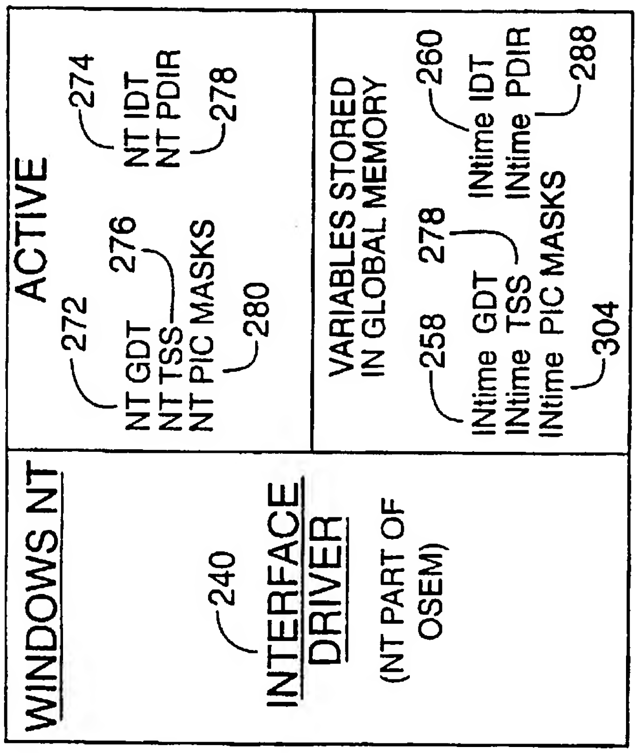


FIG. 12f

FIG. 13

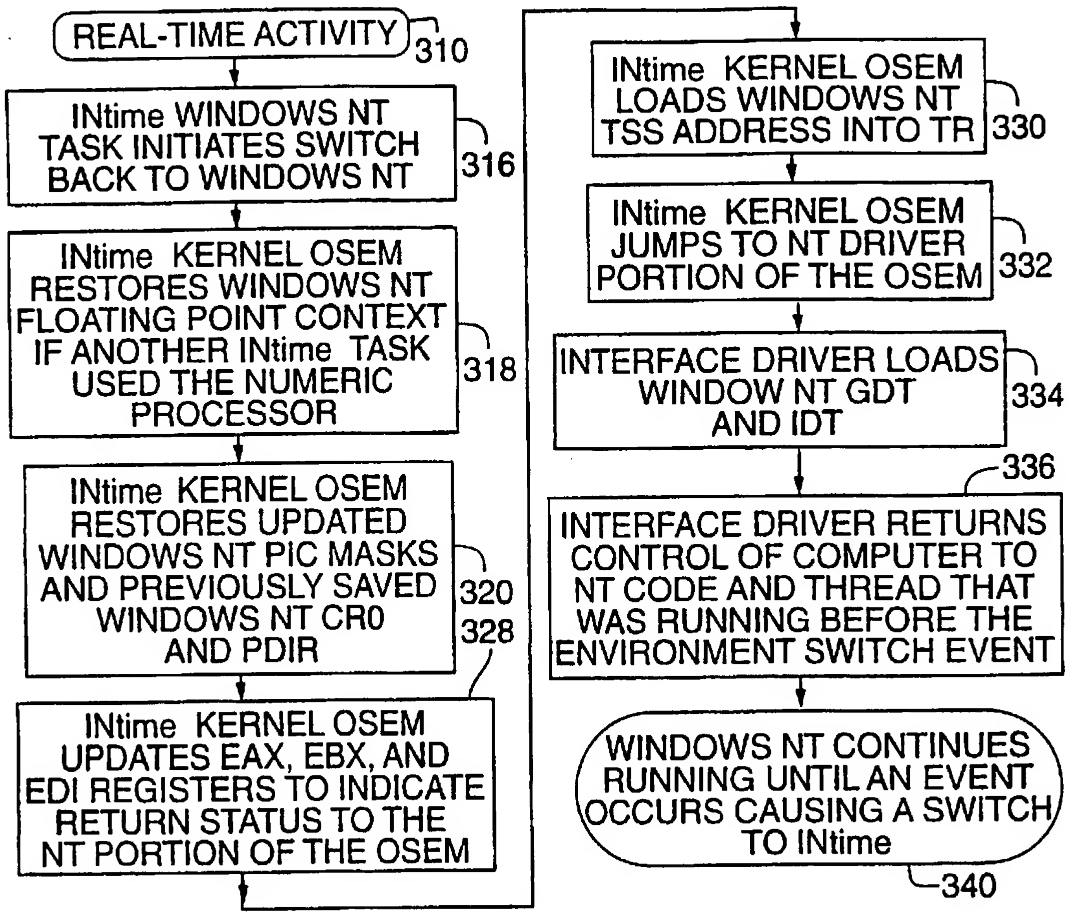
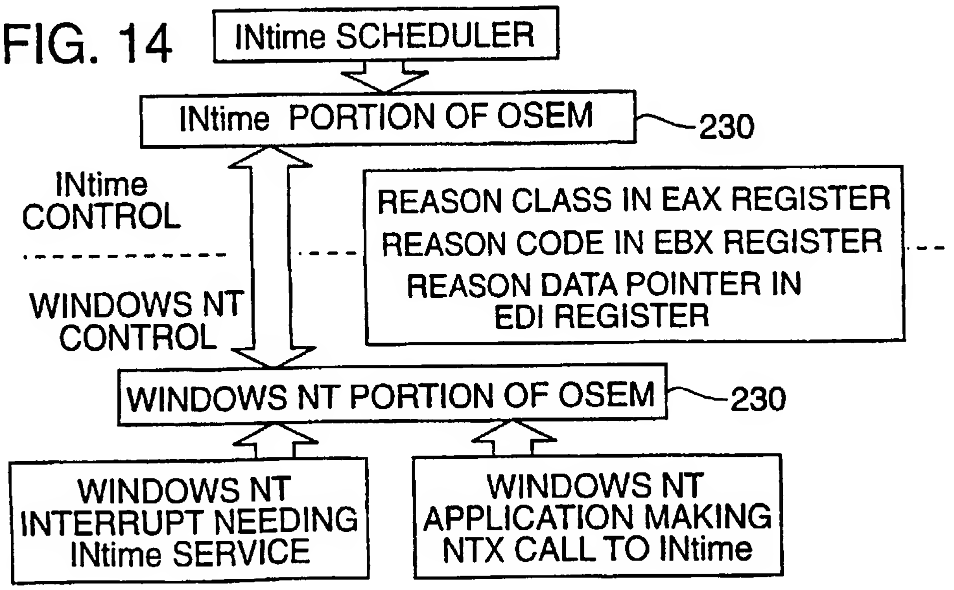

REAL-TIME ACTIVITY
310
INtime WINDOWS NT TASK INITIATES SWITCH BACK TO WINDOWS NT
316
INtime KERNEL OSEM RESTORES WINDOWS NT FLOATING POINT CONTEXT IF ANOTHER INtime TASK USED THE NUMERIC PROCESSOR
318
INtime KERNEL OSEM RESTORES UPDATED WINDOWS NT PIC MASKS AND PREVIOUSLY SAVED WINDOWS NT CR0 AND PDIR
320
328
INtime KERNEL OSEM UPDATES EAX, EBX, AND EDI REGISTERS TO INDICATE RETURN STATUS TO THE NT PORTION OF THE OSEM
INtime KERNEL OSEM LOADS WINDOWS NT TSS ADDRESS INTO TR
330
INtime KERNEL OSEM JUMPS TO NT DRIVER PORTION OF THE OSEM
332
INTERFACE DRIVER LOADS WINDOW NT GDT AND IDT
334
336
INTERFACE DRIVER RETURNS CONTROL OF COMPUTER TO NT CODE AND THREAD THAT WAS RUNNING BEFORE THE ENVIRONMENT SWITCH EVENT
WINDOWS NT CONTINUES RUNNING UNTIL AN EVENT OCCURS CAUSING A SWITCH TO INtime
340

FIG. 14


INtime SCHEDULER
INtime PORTION OF OSEM
230
INtime CONTROL
WINDOWS NT CONTROL
REASON CLASS IN EAX REGISTER
REASON CODE IN EBX REGISTER
REASON DATA POINTER IN EDI REGISTER
WINDOWS NT PORTION OF OSEM
230
WINDOWS NT INTERRUPT NEEDING INtime SERVICE
WINDOWS NT APPLICATION MAKING NTX CALL TO INtime

# INTERNATIONAL SEARCH REPORT

Intern    ıal Application No
PCT/US 97/16511

**A. CLASSIFICATION OF SUBJECT MATTER**
IPC 6    G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)
IPC 6    G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category ° | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| X | WROBEL M: "WINDOWS WINDUSTRIETAUGLICH AUFRUSTEN" ELEKTRONIK, vol. 42, no. 17, 24 August 1993, pages 95-100, XP000387672 see page 95, left-hand column, line 1 - page 99, left-hand column, line 9 --- | 1-13 |
| A | SIWON P: "BETRIEBSARTEN DES 80386" ELEKTRONIK, vol. 38, no. 19, 15 September 1989, pages 53/54, 56-59, XP000054106 see the whole document --- | 1,3-6,9, 11,12 |
| A | US 5 355 498 A (PROVINO JOSEPH E ET AL) 11 October 1994 see column 1, line 48 - line 53 ----- | 8 |

☐ Further documents are listed in the continuation of box C.    ☒ Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier document but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
"&" document member of the same patent family

1

| | |
|---|---|
| Date of the actual completion of the international search<br>14 January 1998 | Date of mailing of the international search report<br>21/01/1998 |
| Name and mailing address of the ISA<br>European Patent Office, P.B. 5818 Patentlaan 2<br>NL – 2280 HV Rijswijk<br>Tel. (+31-70) 340-2040, Tx. 31 651 epo nl,<br>Fax: (+31-70) 340-3016 | Authorized officer<br>Kingma, Y |

# INTERNATIONAL SEARCH REPORT

.ormation on patent family members

Interr nal Application No
PCT/US 97/16511

| Patent document cited in search report | Publication date | Patent family member(s) | Publication date |
|---|---|---|---|
| US 5355498 A | 11-10-94 | NONE | |